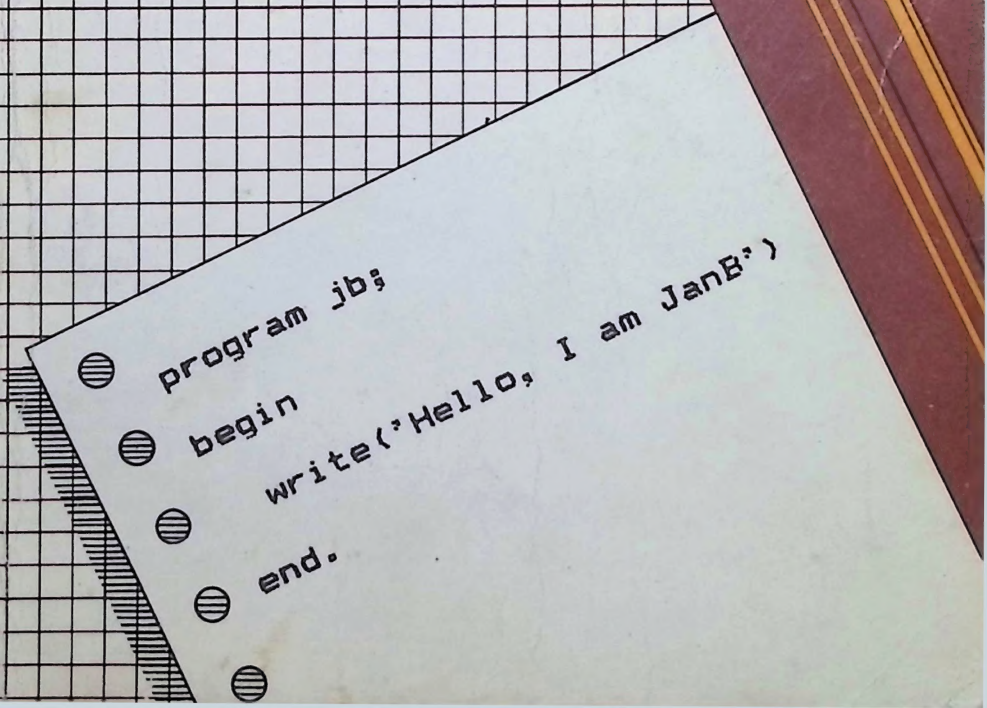
Mikrokomputery

Jan Bielecki

TURBO PASCAL wersja 3.0



Turbo Pascal
wersja 3.0

Dedykuję Kai

Mikrokomputery

Komitet Redakcyjny

Sekretarz Wojciech Cellary
Zuzanna Grzejszczak
Andrzej Kobus
Przewodniczący Romuald Marczyński
Piotr Misiurewicz
Wojciech Nowakowski
Maciej Stolarski
Halina Tempczyk
Józef Winkowski
Jan Zabrodzki

Jan Bielecki

Turbo Pascal

wersja 3.0

Wydawnictwa Naukowo-Techniczne
Warszawa 1987

Opiniodawca *Wojciech Cellary*
Redaktorzy *Hanna Włodarska, Ewa Zdanowicz*
Przygotowała do druku *Hanna Włodarska*
Okładkę projektował *Juliusz Rybicki*

681.3.06

W książce przedstawiono jeden z najważniejszych współczesnych języków programowania – Turbo Pascal. Opis języka wzbogacono licznymi przykładami ilustrującymi poszczególne konstrukcje językowe. Książka jest przeznaczona dla użytkowników mikrokomputerów.

© Copyright by Wydawnictwa Naukowo-Techniczne, Warszawa 1987

All rights reserved
Printed in Poland

ISBN 83-204-0949-7

WNT. Warszawa 1987. Wydanie I.
Nakład 39 700 + 300 egz. Ark. wyd. 10,0. Ark. druk. 11,0
Format B5. Papier offset kl. III. 80 g.
Podpisano do druku w sierpniu 1987 r.
Druk ukończono w październiku 1987 r.
Symbol Et/82362/WNT
Olsztyńskie Zakłady Graficzne. Zam. 1013/87. K-13/222

SPIS TRESCI

1. **Wstęp** / 1

2. Jednostki leksykalne, odstępy, komentarze / 10
2.1. Słowa kluczowe / 11
2.2. Identyfikatory / 12
2.3. Literały / 13
2.4. Komentarze / 15

3. **Standardowe typy skalarne** / 17

4. **Struktura programu** / 20

4.1. Nagłówek programu / 20
4.2. Blok / 21

5. **Wyrażenia** / 27

5.1. Operatory / 27
5.2. Operator zmiany znaku / 28
5.3. Operator negacji / 28
5.4. Operatory typu mnożenia / 29
5.5. Operatory typu dodawania / 31
5.6. Relacje / 33
5.7. Wywołania funkcji / 33

6. **Instrukcje** / 34

6.1. Instrukcja przypisania / 34
6.2. Instrukcja wywołania procedury / 35
6.3. Instrukcja przejścia / 35
6.4. Instrukcja pusta / 36
6.5. Instrukcja grupująca / 36
6.6. Instrukcja warunkowa / 37
6.7. Instrukcja wyboru / 38
6.8. Instrukcje iteracyjne / 39

7. **Typy wyliczeniowe i okrojone** / 42

7.1. Typy wyliczeniowe / 42
7.2. Typy okrojone / 44
7.3. Konwersje typów / 46
7.4. Kontrola poprawności zakresu / 46

8. **Typy łańcuchowe** / 48

8.1. Złączenia łańcuchów / 49
8.2. Relacje / 49
8.3. Przypisania / 50
8.4. Podprogramy / 50
8.5. Typy łańcuchowe i znakowe / 55

9. **Typy tablicowe** / 57

9.1. Tablice wielowymiarowe / 58
9.2. Tablice znakowe / 59
9.3. Przypisania / 60
9.4. Tablice predefiniowane / 61

10. **Typy rekordowe** / 63

10.1. Instrukcja with / 65
10.2. Traktowanie wariantów jako unii / 67
10.3. Przypisania rekordów / 67

11. **Typy mnogościowe** / 69
11.1. Literały mnogościowe / 70
11.2. Wyrażenia / 70
11.3. Przypisania mnogości / 72

12. **Typy plikowe** / 73

12.1. Podprogramy / 74
12.2. Pliki tekstowe / 83
12.3. Operacje na plikach tekstowych / 87
12.4. Pliki bez typu elementów / 99
12.5. Kontrolowanie poprawności operacji wejścia/wyjścia / 102

13. **Typy wskazujące** / 103

14. **Przypisywanie danych początkowych** / 109

15. **Funkcje i procedury** / 114

16. **Podprogramy standardowe** / 123

16.1. Procedury ekranowe / 123
16.2. Procedury specjalne / 125

16.3. Funkcje arytmetyczne / 127
16.4. Funkcje skalarne / 129
16.5. Funkcje do wykonywania konwersji / 130
16.6. Funkcje pomocnicze / 131

17. **Włączanie zbiorów** / 134

18. **Nakładkowanie podprogramów** / 136

19. **Wybrane rozszerzenia implementacyjne** / 140

19.1. Jawne przydzielanie miejsca zmiennym programu / 140
19.2. Dodatkowe funkcje i procedury systemowe / 141
19.3. Wywoływanie podprogramów systemowych (Elwro 800 Jr) / 142
19.4. Programowanie w języku asemblera (Elwro 800 Jr) / 144
19.5. Aktywowanie innych programów / 146

20. **Reprezentowanie danych** / 148
20.1. Dane typu porządkowego /148
20.2. Dane typu rzeczywistego / 149
20.3. Dane typu łańcuchowego / 150
20.4. Dane typu mnogościowego / 150
20.5. Dane wskazujące / 151
20.6. Tablice / 152
20.7. Dane rekordowe / 152

Literatura / 153

Dodatek A. Tablica kodu ASCII / 154

Dodatek B. Wykazy błędów / 156

B.1. Błędy sygnalizowane podczas kompilowania programów / 156
B.2. Błędy sygnalizowane podczas wykonywania programów / 158

Dodatek C. Edytor ekranowy WordStar / 160

C.1. Pojęcia podstawowe / 160
C.2. Wprowadzanie, zmiany i usuwanie porcji tekstu / 161
C.3. Złożone operacje na tekście / 165
C.4. Dyrektywy pomocnicze / 167

Historia języka Pascal datuje się od 1974 roku, kiedy to Niklaus Wirth - jego twórca - oraz Kathleen Jensen opublikowali poprawione opracowanie zawierające opis języka.

Początkowo Pascal służył wyłącznie jako narzędzie do nauczania programowania strukturalnego. Opracowanie tanich i dobrych kompilatorów spowodowało jednak, że język bardzo się rozpowszechnił i należy obecnie do najważniejszych współczesnych języków programowania - zwłaszcza zagadnień o charakterze systemowym.

Jednym z ważnych dialektów standardowego języka Pascal, zgodnym z nim w ponad 90%, jest język Turbo Pascal. Został on opracowany z myślą o zastosowaniach mikrokomputerowych i bezpośrednio po wprowadzeniu na rynek przez firmę Borland odniósł światowy sukces.

Do najważniejszych zalet języka Turbo Pascal należą: mały rozmiar kompilatora, bardzo szybka kompilacja programów, złączenie kompilatora języka z interakcyjnym edytorem ekranowym, sygnalizowanie błędów na poziomie programu źródłowego, obszerna biblioteka podprogramów oraz rozszerzenia w stosunku do języka standardowego ułatwiające programowanie systemowe.

Nazwa Turbo Pascal dotyczy interakcyjnego systemu programowania, składającego się z kompilatora języka Turbo Pascal oraz złączonego z nim edytora ekranowego. Edytor systemu Turbo Pascal jest wzorowany na powszechnie znanym edytorze WordStar, a interakcyjność systemu przejawia się w głównej mierze wygodą edycji oraz sygnalizowaniem błędów wykrytych w programach bezpośrednio w obrębie tekstu źródłowego.

Naturalne i wymagające minimalnej liczby manipulacji przechodzenie między edycją, kompilacją i wykonaniem programu

zapewnia, że uruchamianie programów w systemie Turbo Pascal odbywa się prawie wyłącznie na poziomie źródłowym. Znaczna szybkość kompilowania programów powoduje natomiast, że przejście od programu źródłowego do programu wykonywalnego jest niemal natychmiastowe. Wydatnie skraca to cykl modyfikowania programu i przyspiesza uzyskanie jego wersji końcowej.

W celu zilustrowania zasad posługiwania się systemem Turbo Pascal zostanie rozpatrzony prosty program do wyznaczania objętości kuli o zadanym promieniu.

```
program Volume;
var
  Rad,Vol : real;
begin
  ClrScr;
  GotoXY(10,11);
  Write('radius = ');
  Readln(Rad);
  Vol:=4/3*Pi*Rad*Rad*Rad;
  GotoXY(10,13);
  Writeln('volume = ',Vol);
  repeat until KeyPressed
end.
```

W programie tym występuje predefiniowana nazwa literału *Pi** stanowiącego przybliżenie liczby π oraz nazwy symboliczne *Rad* i *Vol* reprezentujące odpowiednio promień i objętość kuli. Wykonanie procedury *ClrScr* powoduje wyzerowanie ekranu, a wykonanie procedury *GotoXY* powoduje ustawienie kursora w pozycji o podanych współrzędnych. W przypadku wywołania *GotoXY*(10,13) jest to 10 kolumna i 13 wiersz. Zarówno numery kolumn, jak i wierszy są liczone od 1 do wartości maksymalnej. Wykonanie cyklu **repeat until** powoduje wstrzymanie wykonywania programu do chwili wprowadzenia dowolnego znaku z klawiatury. Znak ten pozostaje w buforze wejściowym i może być następnie zinterpretowany przez system Turbo Pascal.

W celu skompilowania i wykonania przytoczonego programu należy w pierwszej kolejności wywołać system Turbo Pascal. Odbywa się to za pomocą wywołania programu Turbo.

---

* Literał jest napisem reprezentującym stałą - obiekt, który podczas wykonywania programu nie ulega zmianie. Wielu autorów miesza te dwa pojęcia i nazywa literały stałymi, a nazwy zmiennych - zmiennymi.

Bezpośrednio po wywołaniu tego programu na ekranie monitora pojawia się komunikat identyfikujący system

```
Turbo Pascal system
Copyright © ... by Borland Inc.
Include error messages (Y/N)?
```

W przytoczonym napisie jest zawarte pytanie, czy do systemu należy dołączyć zbiór tekstów stanowiących komunikaty. Ponieważ komunikaty te zajmują zaledwie 1,5 Kb pamięci, a ich występowanie znacznie ułatwia diagnostykę, zaleca się udzielenie odpowiedzi Y.

W chwilę po udzieleniu odpowiedzi, na ekranie monitora pojawia się menu, w którym wybrane litery poszczególnych słów są wyróżnione przez rozjaśnienie.

```
Logged drive:
Work file:
Main file:

Edit      Compile  Run     Save
Dir       Quit compiler    Options

Text:
Free:
```

Wprowadzenie z klawiatury dyrektywy mającej postać jednej z wyróżnionych liter powoduje stworzenie warunków do interakcyjnej zmiany parametrów systemu.

Dyrektywa L

Wprowadzenie z klawiatury litery L umożliwia zmianę domniemanej stacji dyskowej, którą bezpośrednio po inicjacji jest stacja domniemana w systemie operacyjnym. Jeśli jest np. pożądane wyróżnienie stacji B, to bezpośrednio po zapytaniu

```
New drive:
```

należy wprowadzić z klawiatury parę znaków B: zakończoną znakiem CR (Elwro 800 Jr) albo Enter (IBM PC).

Dyrektywa W

Wprowadzenie z klawiatury dyrektywy W umożliwia określenie nazwy zbioru roboczego zawierającego program źródłowy. Zbiór ten może już istnieć, ale również może być zbiorem,

który ma być dopiero utworzony. Bezpośrednio po zapytaniu

Work file name:

należy wprowadzić tekst określający nazwę zbioru, zakończony znakiem CR (Elwro 800 Jr) albo Enter (IBM PC). Dla rozpatrywanego programu przykładowego mógłby to być np. tekst TB01.PAS.

W systemie CP/M-80 (Elwro 800 Jr) oraz PC DOS (IBM PC) nazwa zbioru może być określona jako dwuczłonowa. Pierwszy człon składa się z nie więcej niż 8 znaków, a drugi - nazywany rozszerzeniem - z nie więcej niż 3 znaków. Człony są oddzielone znakiem . (kropka). Jeśli w odpowiedzi na rozpatrywane zapytanie o nazwę zbioru ograniczyć się do pierwszego członu nazwy, to zostanie domniemana kropka oraz rozszerzenie PAS. Aby uniknąć domniemania rozszerzenia, można ograniczyć się do podania pierwszego członu nazwy oraz kropki.

Należy nadmienić, że jeśli ustalenie nowej nazwy zbioru roboczego następuje w sytuacji gdy nie zapamiętano na dysku poprzedniego zbioru roboczego, to system wyprowadza komunikat zapytujący, czy zbiór poprzedni powinien zostać zapamiętany. Odpowiedzią na takie zapytanie jest jak zwykle Y (yes) albo N (no).

Dyrektywa M

Wprowadzenie z klawiatury dyrektywy M umożliwia określenie nazwy zbioru poddawanego kompilacji. Jeśli nazwa taka nie zostanie ustalona, to kompilacji podlega aktualny plik roboczy.

Zasady określania nazwy zbioru poddawanego kompilacji, nazywanego dalej zbiorem głównym, są takie same jak dla zbioru roboczego. Użycie zbioru głównego jest wygodne wówczas, gdy program źródłowy zawiera dyrektywy włączające w miejscu ich wystąpienia zawartość innych zbiorów. W takim przypadku stwierdzenie błędu kompilacji w jednym ze zbiorów włączonych czyni go zbiorem roboczym, nadającym się do natychmiastowej edycji. Po jej wykonaniu można w prosty sposób ponowić kompilację programu zawartego w zbiorze głównym.

Dyrektywa E

Wprowadzenie z klawiatury dyrektywy E powoduje wywołanie edytora i podjęcie edycji zbioru roboczego. Jeśli do tej chwili nie została określona nazwa zbioru roboczego, to sys-

tem wyprowadza komunikat jak podczas wprowadzania dyrektywy W, a edycję podejmuje dopiero po zidentyfikowaniu zbioru.

Dyrektywa C

Wprowadzenie z klawiatury dyrektywy C powoduje wykonanie kompilacji zbioru głównego. Jeśli do tej chwili nie określono nazwy tego zbioru, to podejmowana jest kompilacja zbioru roboczego. Jeśli określono nazwę zbioru głównego, a poddawano edycji zbiór roboczy, to rozpoczęcie kompilacji zostanie poprzedzone zapamiętaniem zbioru roboczego w pamięci zewnętrznej.

W wyniku wykonania kompilacji powstaje program rezydujący w pamięci operacyjnej lub program umieszczony w zbiorze z rozszerzeniem .COM albo .CHN. Decyzja o sposobie przeprowadzania kompilacji jest podejmowana na podstawie opcji kompilacji ustawianych za pomocą dyrektywy O. Przez domniemanie przyjmuje się, że program wynikowy ma być umieszczony w pamięci operacyjnej.

Jeśli podczas wykonywania kompilacji zostanie wprowadzony z klawiatury dowolny znak, to kompilacja zostanie wstrzymana. Bezpośrednio po udzieleniu odpowiedzi na zapytanie

Abort compilation (Y/N)?

nastąpi zaniechanie albo kontynuowanie kompilacji.

Dyrektywa R

Wprowadzenie z klawiatury dyrektywy R powoduje wykonanie właśnie skompilowanego programu rezydującego w pamięci operacyjnej. Jeśli za pomocą dyrektywy O zostanie ustawiona opcja Com-file, to wykonanie będzie dotyczyć tego programu znajdującego się w pamięci zewnętrznej. Jeśli przed użyciem dyrektywy R nie posłużono się dyrektywą C, to podjęcie wykonywania programu zostanie automatycznie poprzedzone jego kompilacją.

Dyrektywa S

Wprowadzenie z klawiatury dyrektywy S powoduje zapamiętanie zbioru roboczego w pamięci zewnętrznej. Poprzednia wersja tego zbioru zostanie przemianowana w taki sposób, że otrzyma rozszerzenie .BAK.

Dyrektywa X (tylko w CP/M-80)

Wprowadzenie z klawiatury dyrektywy X umożliwia wykonanie dowolnego programu znajdującego się w zbiorze o podanej nazwie. Bezpośrednio po zapytaniu

Program:

należy podać nazwę zbioru z rozszerzeniem .COM zawierającego program wykonywalny.

Dyrektywa D

Wprowadzenie z klawiatury dyrektywy D umożliwia wyprowadzenie nazw zbiorów znajdujących się w katalogu. Bezpośrednio po zapytaniu

Dir mask:

należy podać nazwę zbioru albo nazwę rodziny zbiorów, np. B:*.JB? (* oznacza dowolny dopuszczalny ciąg znaków, a ? - dowolny dopuszczalny znak). Jeśli udzielając odpowiedzi na zapytanie nie podano nazwy stacji dyskowej albo nazwy katalogu, to zostaną one domniemane zgodnie z menu.

Dyrektywa Q

Wprowadzenie z klawiatury dyrektywy Q powoduje zakończenie współpracy z systemem Turbo Pascal i wywołanie systemu operacyjnego. Jeśli przed użyciem dyrektywy Q poddawano edycji zbiór roboczy, to wywołanie systemu operacyjnego zostanie poprzedzone zapytaniem, czy zbioru tego nie należy zapamiętać w pamięci zewnętrznej.

Dyrektywa O

Wprowadzenie z klawiatury dyrektywy O umożliwia określenie opcji kompilacji. Przez domniemanie przyjmuje się, że program wykonywalny zostanie umieszczony w pamięci operacyjnej. Jeśli po użyciu dyrektywy O zostanie użyta poddyrektywa C, to program ten będzie umieszczony w pamięci zewnętrznej jako zbiór z rozszerzeniem .COM. Przywrócenie pierwotnego domniemania można uzyskać za pomocą poddyrektywy M.

Jeśli zostanie użyta poddyrektywa H, to program wykonywalny zostanie umieszczony w pamięci zewnętrznej jako zbiór o nazwie z rozszerzeniem .CHN. Program zawarty w takim zbiorze różni się tym od programu zawartego w zbiorze o nazwie z roz-

szerzeniem .COM, że nie zawiera biblioteki podprogramów standardowych i musi być wywoływany z innego programu za pomocą specjalnej procedury *Chain*.

Posłużenie się poddyrektywą P umożliwia określenie parametrów programu wykonywalnego umieszczonego w pamięci operacyjnej. Zostaną one przekazane programowi dokładnie w taki sposób, w jaki są przekazywane parametry programom wywoływanym z pamięci zewnętrznej.

Użycie poddyrektywy F umożliwia zlokalizowanie błędu wykonywania programu umieszczonego w zbiorze o nazwie z rozszerzeniem .COM albo .CHN. W odróżnieniu od programów wykonywalnych umieszczanych w pamięci operacyjnej, których błędy wykonywania są sygnalizowane przez wskazanie miejsca błędu w tekście źródłowym, błędy wykonywania programów umieszczanych w pamięci zewnętrznej są sygnalizowane w sposób zakodowany, przez podanie numeru błędu i stanu licznika instrukcji. Jeśli po zapytaniu wywołanym użyciem poddyrektywy F zostanie podany ten właśnie stan licznika instrukcji, to automatycznie zostanie wskazane miejsce w programie źródłowym, gdzie wystąpił błąd wykonywania.

Ostatnią poddyrektywą dyrektywy O jest poddyrektywa Q. Jej wykonanie kończy wykonywanie dyrektywy O i powoduje przywrócenie głównego menu.

Jak wynika z przytoczonego opisu, posługiwanie się dyrektywami i poddyrektywami systemu Turbo Pascal jest w miarę nieskomplikowane. Równie proste jest wprowadzanie i poprawianie programu źródłowego, następujące po użyciu dyrektywy E albo automatycznie, po zlokalizowaniu błędu podczas kompilowania programu.

Zasady posługiwania się edytorem systemu Turbo Pascal wynikają z jego podobieństwa do procesora tekstów WordStar, na którym jest on wzorowany. Odsyłając do dodatku wszystkich zainteresowanych pełnymi możliwościami tego edytora, ograniczymy się do przytoczenia minimalnego zestawu dyrektyw umożliwiających obróbkę programów źródłowych.

Na wstępie należy zauważyć, że jeśli ktoś nie popełnia błędów, to bezpośrednio po wywołaniu edytora za pomocą dyrektywy E może wprowadzić tekst programu jako ciąg wierszy zakończonych znakami CR (Elwro 800 Jr) albo Enter (IBM PC), a następnie posłużyć się dyrektywą KD kończącą edycję i przywracającą główne menu systemu.

Dyrektywa KD składa się z liter K i D i podobnie jak pozostałe dyrektywy edytora systemu Turbo Pascal wymaga posłużenia się klawiszem Ctrl. Klawisz ten powinien być wciśnięty podczas wprowadzania litery K. Będzie to oznaczane skrótowo za pomocą opisu Ctrl-K-D albo ^KD.

W obrębie tekstu wyświetlanego na ekranie monitora jeden ze znaków jest wyróżniony za pomocą kursora. Operacje na tekście są zawsze wykonywane w odniesieniu do tego znaku, zawierającego go słowa albo wiersza.

Przesunięcie kursora o jedną pozycję w lewo, w prawo, w górę lub w dół odbywa się za pomocą dyrektyw Ctrl-S, Ctrl-D, Ctrl-E lub Ctrl-X. Klawisze S, D, E i X tych dyrektyw tworzą romb

E
S D
X

a jak łatwo zapamiętać, dyrektywa Ctrl-E przesuwa kursor o jedną pozycję w górę, dyrektywa Ctrl-X - o jedną pozycję w dół, dyrektywa Ctrl-S - o jedną pozycję w lewo, a dyrektywa Ctrl-D - o jedną pozycję w prawo. Dzięki takiemu rozwiązaniu o funkcji dyrektywy decyduje nie mnemonika nazwy jej klawisza, lecz położenie klawisza w przytoczonym układzie czterokierunkowym.

Usuwanie znaków tekstu odbywa się za pomocą klawisza ← (Backspace). Każdorazowe naciśnięcie tego klawisza powoduje usunięcie znaku znajdującego się z lewej strony znaku wyróżnionego przez kursor. Wstawianie nowych znaków między słowami lub w obrębie słów tekstu nie wymaga specjalnych zabiegów. Każdy z takich znaków jest wstawiany bezpośrednio przed znakiem wyróżnionym przez kursor, a pozostałe znaki wiersza zostają przesunięte o jedną pozycję w prawo. Usuwanie całych wierszy odbywa się za pomocą dyrektywy Ctrl-Y, usuwanie słów - za pomocą dyrektywy Ctrl-T, a wstawianie nowych wierszy - za pomocą dyrektywy Ctrl-N. Wiele innych dyrektyw i możliwości edytora opisano w dodatku.

Podsumowując to, co przedstawiono w niniejszym rozdziale, można podać, że uzyskanie wykonywalnego programu przedstawionego na wstępie wymaga wykonania następujących czynności:

- Wywołania systemu Turbo Pascal
  TURBO
- Dołączenia do kompilatora zbioru komunikatów
  Y
- Określenia nazwy zbioru roboczego
  W
- Zainicjowania wprowadzenia programu źródłowego
  E
- Polecenia wykonania kompilacji
  C

- Polecenia wykonania programu
  R

Po wykonaniu tych czynności można powrócić do systemu operacyjnego, posługując się w tym celu dyrektywą Q.

W przypadku gdyby było pożądane przechowanie programu wykonywalnego jako zbioru z rozszerzeniem .COM, należałoby przed wykonaniem kompilacji posłużyć się dyrektywą O, a następnie poddyrektywami C i Q. Po wykonaniu kompilacji i powrocie do systemu operacyjnego wspomniany program mógłby być wykonywany już bez udziału systemu Turbo Pascal.

2. JEDNOSTKI LEKSYKALNE, ODSTĘPY, KOMENTARZE

Przystępując do szczegółowego omówienia języka programowania Turbo Pascal, należy zacząć od jego *jednostek leksykalnych*, którymi - jak w każdym języku programowania - są *słowa kluczowe*, *identyfikatory*, *literały* i *separatory*. Spacje, znaki tabulacji, znaki przejścia do nowego wiersza i komentarze nie stanowią jednostek leksykalnych. Każdy ciąg takich znaków i komentarzy jest traktowany tak jak pojedyncza spacja i będzie krótko nazywany *odstępem*. Użycie odstępu jest niezbędne jedynie wówczas, gdy w programie źródłowym sąsiadują ze sobą identyfikatory albo słowa kluczowe.

Zgodnie z przytoczoną klasyfikacją każdy program zapisany w języku Turbo Pascal składa się z jednostek leksykalnych i odstępów. Wyodrębnienie tych obiektów odbywa się podczas analizowania programu, dokonywanego w naturalnym porządku, tj. od lewej do prawej i od góry do dołu. Za jednostkę leksykalną jest uznawany w takim przypadku najdłuższy ciąg znaków nie zawierający odstępów, który może uchodzić za jednostkę leksykalną.

Jednostki leksykalne i komentarze są tworzone z *symboli podstawowych*, do których zalicza się:

- małe i duże litery alfabetu angielskiego, wraz ze znakiem podkreślenia

  _, a, b, c, d, e, f, g, h, i, j, k, l, m,
  n, o, p, q, r, s, t, u, v, w, x, y, z,
  A, B, C, D, E, F, G, H, I, J, K, L, M,
  N, O, P, Q, R, S, T, U, V, W, X, Y, Z

- cyfry dziesiętne

  0, 1, 2, 3, 4, 5, 6, 7, 8, 9

- znaki specjalne

  + - * / = { }
  ( ) [ ] < > ^
  . , : ; ' # $

- symbole złożone

  przypisania: :=
  relacje: <> <= >=
  zakresy: ..
  nawiasy: (* *) (. .)
  (nawiasy z gwiazdkami pełnią rolę nawiasów klamrowych, a nawiasy z kropkami - nawiasów kwadratowych).

Litery duże i małe nie są odróżniane. Powoduje to m.in., że identyfikatory takie jak np. *filename* i *FileName* są uznawane za identyczne.

2.1. SŁOWA KLUCZOWE

Słowem kluczowym jest spójny ciąg liter tworzący jedno z wymienionych niżej słów

absolute	external	nil	shl
and	file	not	shr
array	for	of	string
begin	forward	or	then
case	function	overlay	to
const	goto	packed	type
div	if	procedure	until
do	in	program	var
downto	inline	record	while
else	label	repeat	with
end	mod	set	xor

Przykład. Najkrótszy program w jezyku Turbo Pascal

```
begin
end.
```

- Program składa się z dwóch słów kluczowych i separatora.
- Wykonanie tego programu nie wywołuje żadnych skutków. ■

2.2. IDENTYFIKATORY

Identyfikatorem jest ciąg literowo-cyfrowy, rozpoczynający się od litery, taki który nie jest słowem kluczowym. "Literą" użytą w identyfikatorze może być dowolna duża albo mała litera alfabetu angielskiego oraz znak podkreślenia. Liczba znaków identyfikatora nie może przekroczyć 127. Wszystkie jego znaki są istotne.

Wiele identyfikatorów ma predefiniowane znaczenie i służy do oznaczania literałów, funkcji i procedur. Do tej grupy należą m.in. następujące identyfikatory:

Abs	*Delete*	*KeyPressed*	*Read*
Addr	*Dispose*	*Length*	*Readln*
ArcTan	*Eof*	*Ln*	*Real*
Assign	*Eoln*	*Lo*	*Release*
Bdos	*Erase*	*LowVideo*	*Rename*
Bios	*Execute*	*Lst*	*Reset*
BiosHL	*Exit*	*Mark*	*Rewrite*
BlockRead	*Exp*	*MaxAvail*	*Round*
BlockWrite	*False*	*MaxInt*	*Seek*
Boolean	*FilePos*	*Mem*	*Sin*
BufLen	*FileSize*	*MemAvail*	*SizeOf*
Byte	*FillChar*	*Move*	*SkpEof*
Chain	*Flush*	*New*	*SkpEoln*
Char	*Frac*	*NormVideo*	*Sqr*
Chr	*FreeMem*	*Odd*	*Sqrt*
Close	*GetMem*	*Ord*	*Str*
ClrEol	*GotoXY*	*Output*	*Succ*
ClrScr	*Halt*	*OvrDrive*	*Swap*
Con	*HeapPtr*	*ParamCount*	*Text*
Concat	*Hi*	*ParamStr*	*Trm*
Copy	*Input*	*Pi*	*True*
Cos	*InsLine*	*Port*	*Trunc*
CrtExit	*Int*	*Pos*	*UpCase*
CrtInit	*Integer*	*Pred*	*Val*
DelLine	*IOresult*	*Ptr*	*Write*
Delay	*Kbd*	*Random*	*Writeln*

Jeśli w programie zostanie zadeklarowany identyfikator występujący na przytoczonej liście, to w zasięgu tej deklaracji zostanie przesłonięte pierwotne znaczenie danego identyfikatora.

Przykład. Przesłonięcie znaczenia identyfikatora

```
program jb;
const
  false = true;
begin
  Writeln(false)
end.
```

● Wykonanie programu powoduje wyprowadzenie napisu *true*.

● Gdyby z programu usunięto tekst między pierwszym średnikiem a słowem **begin** (wyłącznie), to wykonanie programu spowodowałoby wyprowadzenie napisu *false*.

2.3. LITERAŁY

Literały dzielą się na *arytmetyczne*, *znakowe*, *łańcuchowe* i *logiczne*. Literały arytmetyczne dzielą się na całkowite i rzeczywiste.

Literały całkowite

Podstawowy *literał całkowity* składa się z ciągu cyfr dziesiętnych, który może być poprzedzony znakiem + (plus) lub - (minus). Wartość liczbowa literału całkowitego musi mieścić się w przedziale domkniętym [-32768,32767]. Literał całkowity, którego wartość mieści się w przedziale domkniętym [0,255], będzie nazywany *literałem bajtowym*.

W języku Turbo Pascal zapewniono możliwość przedstawiania literałów całkowitych i bajtowych za pomocą liczb szesnastkowych. W takim przypadku literał składa się ze znaku $ (dolar), po którym następują cyfry szesnastkowe. Przed znakiem $ może występować znak + (plus) albo - (minus).

Przykład. Kilka poprawnych i niepoprawnych literałów całkowitych

```
Literały całkowite:     23
                        -23
                        002
                        $ABC
                        -$2B
Napisy, które nie są literałami całkowitymi:
  2B6   -   znak B nie jest cyfrą dziesiętną
  $3G   -   znak G nie jest cyfrą szesnastkową
  2.3   -   znak . nie jest cyfrą                ■
```

Literały rzeczywiste

Literał rzeczywisty składa się z następujących po sobie: części całkowitej, kropki, części ułamkowej, małej lub dużej litery E oraz wykładnika. Część całkowita, ułamkowa i wykładnik składają się z ciągów cyfr dziesiętnych, a część całkowitą i cyfry wykładnika może poprzedzać znak + (plus) lub - (minus). Część ułamkową wraz z kropką albo wykładnik wraz z literą E można pominąć. Zezwala się również, aby część ułamkowa literału była pusta.

Przykład. Kilka poprawnych i niepoprawnych literałów rzeczywistych

```
Literały rzeczywiste:      -2.3
                           2.5e2
                           4.E-3
                           2E3
Napisy, które nie są literałami rzeczywistymi:
   3e2.0  -  kropka w wykładniku
   .25    -  brak części całkowitej
   2.3D2  -  błędny wykładnik                    ■
```

Literały łańcuchowe

Literały łańcuchowe reprezentują ciągi znaków. Jeśli literał łańcuchowy reprezentuje jeden znak, to jest nazywany *literałem znakowym*. W ogólnym przypadku literał łańcuchowy składa się z sekwencji znaków zawartych między parą apostrofów, znaków sterujących i znaków wyrażonych dziesiętnie i szesnastkowo. W sekwencji zawartej między parą apostrofów mogą występować dowolne znaki widoczne wraz ze spacją, a znak ' (apostrof) jest reprezentowany przez parę sąsiadujących apostrofów. Znaki sterujące są przedstawiane za pomocą pary znaków, z których pierwszym jest ^ (caret), a drugi jest znakiem widocznym. Reprezentacja tak przedstawionego znaku sterującego jest identyczna z reprezentacją znaku uzyskanego przez jednoczesne naciśnięcie klawisza Ctrl i danego znaku widocznego. Znaki wyrażone dziesiętnie składają się ze znaku # (hash), po którym następuje dziesiętny kod znaku, a znaki wyrażone szesnastkowo składają się z następujących bezpośrednio po sobie: znaku # (hash), znaku $ (dolar) i szesnastkowego kodu znaku.

Przykład. Kilka poprawnych i niepoprawnych literałów łańcuchowych

```
Literały łańcuchowe:  'jb'        (dwa znaki)
                      ''''        (jeden znak)
                      ^G          (jeden znak)
                      #65         (jeden znak)
                      #$41        (jeden znak)
                      ^G'jb'      (trzy znaki)
                      ' '         (dwa znaki spacji)
                      ^G^G^G'hello'#13#10'jan'
                                  (13 znaków)
Napisy, które nie są literałami łańcuchowymi:
  '''   - brak apostrofu
 #256   - kod większy niż 255
 #$4G   - G nie jest cyfrą szesnastkową
```

■

Literały logiczne

Literały logiczne mają postać napisów *true* i *false*. Pierwszy z nich reprezentuje daną logiczną "prawda", a drugi - daną logiczną "fałsz".

Przykład. Kilka poprawnych i niepoprawnych literałów logicznych

```
Literały logiczne:  false
                    FALSE
                    True
Napisy, które nie są literałami logicznymi:
 prawda  - błędny napis
 fałsz   - błędny napis
```

■

2.4. KOMENTARZE

Komentarzem jest napis rozpoczynający się od nawiasu klamrowego otwierającego, zakończony najbliższym nawiasem klamrowym zamykającym. Rolę nawiasów klamrowych: { (otwierającego) i } (zamykającego) mogą pełnić symbole złożone (* i *). Komentarze ograniczone znakami { i } mogą być zagnieżdżone w komentarzach ograniczonych symbolami (* i *), a komentarze ograniczone symbolami (* i *) mogą być zagnieżdżone w komentarzach ograniczonych znakami { i }.

Jeśli bezpośrednio po znaku albo symbolu rozpoczynającym komentarz występuje znak $ (dolar), to komentarz taki zo-

staje uznany za dyrektywę kompilatora. Dyrektywa taka nie może znajdować się w komentarzu zagnieżdżonym.

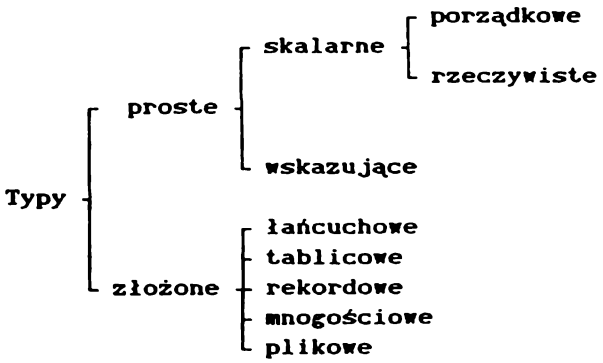
Przykład. Kilka poprawnych i niepoprawnych komentarzy i dyrektyw kompilatora

```
Komentarze:    (* data *)
               { data }
               (* out {file}*)
Dyrektywy:     (*$i include.set *)
               {$v,b+}
Napisy, które nie są komentarzami:
 (* inside }    - niepoprawne ograniczenie komentarza
 {{inside}}     - niepoprawne zagnieżdżenie
Napisy, które nie są dyrektywami kompilatora:
 { $v- }        - spacja między { i $
 {$ v- }        - spacja między $ i v-
```

■

Pojęcie typu wiąże się z ustalonym zbiorem danych. W tym sensie zmienna jest pewnego typu, jeśli można jej przypisywać dane należące do zbioru określającego ten typ. W języku Turbo Pascal wymaga się jawnego określenia typu każdej ze zmiennych programu.

Związanie nazwy zmiennej z wybranym typem odbywa się w *deklaracji zmiennej*. *Typy zmiennych*, sklasyfikowane jak na rys. 3.1, dzielą się na *proste* i *złożone*. Typami prostymi są typy *skalarne* i *wskazujące*, a typami złożonymi - typy *tablicowe*, *rekordowe*, *mnogościowe* i *plikowe*. Wśród typów skalarnych można wyróżnić standardowe typy skalarne, takie jak: *integer* (całkowity), *byte* (bajtowy), *char* (znakowy), *boolean* (logiczny) i *real* (rzeczywisty).Pierwsze cztery będą również nazywane typami porządkowymi. Mają one tę właściwość, że ich elementy są nie tylko uporządkowane, ale również tworzą zbiór przeliczalny.



Rys.3.1. Klasyfikacja typów

Standardowe typy skalarne są predefiniowane, tzn. nie wymagają definiowania. Zasięgiem ich niejawnych definicji jest cały program, z wykluczeniem tych jego fragmentów, w których wprowadzono jawne deklaracje przesłaniające.

Typ integer

Dana typu *integer* jest elementem podzbioru liczb całkowitych. Wartości tych liczb należą do przedziału domkniętego [-32768,32767]. Każda z danych typu *integer* jest reprezentowana w 2 bajtach pamięci.

Podczas wykonywania działań arytmetycznych na danych typu *integer* nie kontroluje się występowania *nadmiaru*. Z tego względu warunkiem uzyskania poprawnego rezultatu operacji na danych typu *integer* jest przynależność argumentów i wszystkich wyników pośrednich do wymienionego wyżej przedziału.

Typ byte

Dana typu *byte* jest elementem podzbioru liczb całkowitych. Wartości danych typu *byte* należą do przedziału domkniętego [0,255]. W prawie każdym miejscu programu, w którym występuje odwołanie do danej bajtowej, może wystąpić odwołanie do danej typu *integer* i odwrotnie. Jedyny wyjątek od tej zasady dotyczy skojarzeń parametrów i argumentów procedur i funkcji. W tym przypadku wymaga się pełnej zgodności typów danych. Każda z danych typu *byte* jest reprezentowana w 1 bajcie pamięci.

Typ char

Dana typu *char* jest elementem zbioru znaków. Każda z danych typu *char* jest reprezentowana w 1 bajcie pamięci.

Typ boolean

Dana typu *boolean* jest elementem dwuelementowego zbioru danych logicznych. Do oznaczenia tych danych służą standardowe identyfikatory *true* i *false*. Każda z danych typu *boolean* jest reprezentowana w 1 bajcie pamięci.

Typ real

Dana typu *real* jest elementem zbioru liczb rzeczywistych. Do zbioru tego należy liczba 0 oraz liczby dodatnie i ujemne, których wartości bezwzględne należą do przedziału $[10^{-38},10^{38}]$. Każda z danych typu *real* jest reprezentowana w 6 bajtach pamięci, z dokładnością do ok. 11 cyfr znaczących.

Program napisany w języku Turbo Pascal składa się z *nagłówka programu*, *bloku* i znaku . (kropka). W nagłówku jest określona nazwa programu oraz ujęty w nawiasy okrągłe wykaz identyfikatorów, opisujący sposób komunikowania się programu z otoczeniem. Blok składa się z części deklaracyjnej i wykonawczej. Zarówno nagłówek, jak i część deklaracyjna bloku mogą być pominięte.

4.1. NAGŁOWEK PROGRAMU

Jeśli program zaczyna się od nagłówka, to występuje w nim przynajmniej nazwa programu. Po nazwie tej może wystąpić - ujęta w nawiasy okrągłe - lista parametrów programu. Lista ta wyszczególnia identyfikatory plików, za pośrednictwem których program komunikuje się z otoczeniem. Liście tej nie jest nadawana żadna interpretacja i z tego względu lista ta (wraz z otaczającymi ją nawiasami) jest pomijana.

Składnia

nagłówek-programu:
program *nazwa-programu* (*lista-nazw-plików*) ;
nazwa-programu:
identyfikator
nazwa-pliku:
identyfikator

Przykłady

```
program Volume;
program Lister(Output);
program Copier(Input,Output);
```

4.2. BLOK

Blok składa się z części deklaracyjnej i części wykonawczej. W części deklaracyjnej znajdują się definicje typów, definicje nazw literałów, definicje i deklaracje podprogramów, a także deklaracje etykiet i zmiennych. Wszystkie te deklaracje i definicje mogą być w dowolnym porządku, a poszczególne grupy deklaracji i definicji mogą się przeplatać. Należy rozumieć to w taki sposób, że np. po definicjach typów mogą następować deklaracje zmiennych, a po nich ponownie definicje typów i znowu deklaracje zmiennych. Część wykonawcza składa się z instrukcji grupującej, zawierającej instrukcje programu.

Składnia

blok:
część-deklaracyjna część-wykonawcza
część-deklaracyjna:
wykaz-deklaracji-i-definicji
część-wykonawcza:
instrukcja-grupująca
deklaracje-i-definicje:
deklaracje-etykiet
definicje-nazw-literałów
definicje-typów
deklaracje-zmiennych
definicje-i-deklaracje-podprogramów
inicjacje

Deklaracje etykiet

Każda instrukcja czynna programu może być poprzedzona *etykietą* umożliwiającą odwołanie się do niej w instrukcji **goto**. Etykieta składa się z identyfikatora albo z ciągu cyfr. Bezpośrednio po etykiecie następuje znak : (dwukropek) oddzielający ją od innej etykiety albo instrukcji. Zanim etykieta zostanie użyta w programie, jej nazwa musi być zadeklarowana. Deklaracje-etykiet składają się ze słowa kluczowego **label**, po którym następuje lista napisów identycznych z nazwami etykiet, zakończona średnikiem.

Składnia

deklaracje-etykiet:
label *lista-etykiet*;

Przykłady

```
label 10, abort, quit;
label 99999;
```

■

Definicje nazw literałów

W celu zwiększenia czytelności programu pożądane jest niekiedy zastąpienie literałów odpowiednio dobranymi identyfikatorami reprezentującymi te literały. W takich przypadkach każde wystąpienie identyfikatora jest równoważne wystąpieniu związanego z nim literału. W języku Turbo Pascal związanie identyfikatorów z literałami odbywa się podczas interpretowania *definicji nazw literałów*. Każda z definicji nazw literałów składa się ze słowa kluczowego **const**, po którym następuje sekwencja przypisań. Każde z przypisań składa się z identyfikatora, bezpośrednio po którym następuje znak = (równa się), wyrażenie stałe oraz znak ; (średnik). Wyrażeniem stałym może być liczba, literał łańcuchowy, nazwa literału albo nazwa literału poprzedzona znakiem - (minus). Predefiniowanymi nazwami literałów są

Pi - typu *real*, reprezentujące literał 3.1415926536
MaxInt - typu *integer*, reprezentujące literał 32767

Składnia

definicje-nazw-literałów:
  const *sekwencja-przypisań*
przypisanie:
  identyfikator = *liczba* ;
  identyfikator = *literał-tekstowy* ;
  identyfikator = *nazwa-literału* ;
  identyfikator = *znak-nazwa-literału* ;

Przykłady

```
const e = 2.718282;
      g = 9.81;
const TwoChar = '2';
      Two     = 2;
      Minus2  = -Two;
const FirstName = 'janek';
```

■

Definicje typów

Podczas deklarowania zmiennych programu następuje określenie ich typu. *Typ zmiennej* może należeć do zbioru predefiniowanych typów standardowych, takich jak np. *boolean* lub

real, ale może także zostać zdefiniowany odrębnie w *definicji typu*. W ogólnym przypadku definicje-typów składają się ze słowa kluczowego **type**, po którym następuje sekwencja definicji typów. Każda z definicji typu składa się z identyfikatora typu definiowanego, znaku = (równa się), opisu typu i znaku ; (średnik).

Składnia

definicje-typów:
type *sekwencja-definicji-typu*
definicja-typu:
typ-definiowany = *opis typu* ;
typ-definiowany:
identyfikator
opis-typu:
opis-typu-standardowego
opis-typu-definiowanego
opis-typu-standardowego:
identyfikator-typu-standardowego
opis-typu-definiowanego:
opis-typu-wyliczeniowego
opis-typu-okrojonego
opis typu-łańcuchowego
opis-typu-tablicowego
opis-typu-rekordowego
opis-typu-mnogościowego
opis-typu-plikowego
opis-typu-wskazującego

Przykłady

```
type
  Number = integer;
  Color = (Red,Green,Blue,Orange);
  RGB = Red..Blue;
  Name = string[20];
  Matrix = array[1..5,1..5] of real;
  Person = record
             FirstName : string[6];
             LastName : Name;
             Salary : real
           end;
  Digits = set of 0..9;
  DataFile = file of Person;
  Ref = Matrix;
```

Deklaracje zmiennych

Każda zmienna programu, zarówno prosta jak i złożona, musi być zadeklarowana przed jej użyciem. Deklaracje zmiennych składają się ze słowa kluczowego var, po którym następuje wykaz deklaracji. *Elementem wykazu deklaracji* jest sekwencja napisów składająca się z listy identyfikatorów, znaku : (dwukropek), oznaczenia typu i znaku ; (średnik). Zinterpretowanie każdej z deklaracji powoduje utworzenie zmiennych podanego typu.

Składnia

deklaracje-zmiennych:
var *sekwencja-deklaracji*
deklaracja:
lista-identyfikatorów : *oznaczenie-typu* ;

Przykłady

```
var
  Length, Area, Volume : real;
  Count : integer;
  Buffer : array[0..255] of byte;
  Rejected : boolean;
```

■

Zakresem deklaracji zmiennej jest blok, w którym zmienna ta została zadeklarowana. Jeśli blok ten zawiera inne bloki, w których występują deklaracje zmiennych oznaczone takim samym identyfikatorem, to zasięg deklaracji jest mniejszy niż zakres i nie obejmuje bloków wewnętrznych, w których wystąpiły deklaracje przesłaniające. Zakres deklaracji zmiennej rozpoczyna się nie od początku bloku, lecz od miejsca zadeklarowania jej identyfikatora.

Przykład. Zakres i zasięg deklaracji

```
                                  zakres      zasięg
program Tricky;                     1           1
var                                 1           1
  Toggle : false1..true;          1       1
  false2 : integer;                 1 2           2
begin                               1 2           2
  false := 5;                       1 2           2
  Writeln(false)                    1 2           2
end.
```

● Zasięgiem predefiniowanego identyfikatora *false* jest jego zakres pomniejszony o zakres jawnie zadeklarowanego identyfikatora *false* typu *integer*.

● Zakres i zasięg zadeklarowanego jawnie identyfikatora *false* jest taki sam.

● Ponieważ wywołanie procedury *Writeln* występuje w zasięgu jawnie zadeklarowanego identyfikatora *false*, wykonanie programu powoduje wyprowadzenie liczby 5. ■

Deklaracje i definicje podprogramów

Podprogramy dzielą się na *funkcje* i *procedury*. Definicja podprogramu określa czynności, jakie zostaną wykonane bezpośrednio po wywołaniu podprogramu. W przypadku funkcji określa również typ rezultatu funkcji udostępnionego w miejscu jej wywołania.

Użycie deklaracji podprogramu jest niezbędne jedynie wówczas, gdy nie jest możliwe takie uporządkowanie definicji podprogramów, aby odwołania do podprogramów występowały w zasięgu ich definicji.

Przykład. Wzajemne wywołania procedur

```
program jb;
  ...
  procedure second; forward;
  procedure first;
  begin
    ...
    second
  end;
  procedure second;
  begin
    ...
    first
  end;
begin
  ...
end.
```

● Ponieważ w definicji procedury *first*

```
procedure first;
begin
  ...
second
end;
```

występuje wywołanie procedury *second*, a w definicji procedury *second* występuje wywołanie procedury *first*, konieczne było użycie deklaracji procedury *second* o postaci

```
procedure second; forward;
```

● W odróżnieniu od definicji deklaracja służy jedynie zaprezentowaniu identyfikatora podprogramu, a jeśli podprogram ma parametry, to także jego parametrów. ■

5. WYRAŻENIA

Wyrażenia są napisami określającymi czynności, jakie mają być wykonane w celu uzyskania danej. Wykonanie tych czynności będzie nazywane *opracowaniem wyrażenia*. Ponieważ rezultatem opracowania wyrażenia jest dana, samo wyrażenie można uważać za napis reprezentujący daną. W niniejszym rozdziale zostaną opisane zasady opracowywania wyrażeń, których elementy składowe reprezentują dane typu *integer*, *real*, *char* i *boolean* . Opisy wyrażeń zawierających odwołania do danych typów definiowanych zostaną przedstawione podczas omawiania tych typów.

5.1. OPERATORY

Operatory dzielą się na *jednoargumentowe* i *dwuargumentowe*. Innym kryterium ich podziału jest *priorytet*. W kolejności malejącego priorytetu ogół operatorów można podzielić na

a) jednoargumentowy operator zmiany znaku
   -
b) jednoargumentowy operator negacji
   not
c) dwuargumentowe operatory typu mnożenia
   *** / div mod and shl shr**
d) dwuargumentowe operatory typu dodawania
   + - or xor
e) dwuargumentowe operatory relacyjne
   = <> < > <= >= in

Jeśli pewnego podwyrażenia dotyczą dwa operatory o równym priorytecie, to stosowane operacje są wykonywane od lewej do prawej. Podwyrażenia ujęte w nawiasy okrągłe są opracowy-

wane w pierwszej kolejności. Z tego względu nawiasy okrągłe mogą być traktowane jak jednoargumentowy operator o najwyższym priorytecie.

Jeśli argumentami operacji typu mnożenia i dodawania są podwyrażenia reprezentujące dane typu *integer* lub *real*, to rezultat operacji jest typu *integer* tylko wtedy, gdy oba argumenty są typu *integer*, a operacją nie jest dzielenie. W pozostałych przypadkach rezultat operacji jest typu *real*.

Przykład. Wyznaczanie danej reprezentowanej przez wyrażenie

```
2 + 3 / 4
```

- Ponieważ priorytet dzielenia jest wyższy niż priorytet dodawania, przytoczone wyrażenie jest traktowane tak jak wyrażenie

```
2 + (3 / 4)
```

a nie jak wyrażenie

```
(2 + 3) / 4
```

- Rezultatem opracowania wyrażenia 3/4 jest dana typu *real* o wartości równej - w przybliżeniu - wartości literału 0.75
- Przytoczone wyrażenie reprezentuje daną typu *real* o wartości bliskiej wartości danej reprezentowanej przez literał 2.75 . ■

5.2. OPERATOR ZMIANY ZNAKU

Wykonanie jednoargumentowej operacji - (minus) powoduje zmianę znaku danej na przeciwny. Jeśli dana ma wartość 0, to rezultatem operacji jest także dana o wartości 0. Argumentami operacji - (minus) mogą być tylko dane typu *real* i *integer*.

Przykłady

Wyrażenie	Rezultat
-(-4)	4
-3/2	-1.5

■

5.3. OPERATOR NEGACJI

Wykonanie jednoargumentowej operacji **not** może dotyczyć jedynie danych typu *boolean* i *integer*. W pierwszym przypadku dana o wartości *false* zostanie przekształcona w daną o wartości *true* i odwrotnie. W drugim, każdy bit reprezentacji danej zostanie zmieniony na przeciwny.

Przykłady

Wyrażenie	Rezultat
not *false*	*true*
not *true*	*false*
not 0	-1
not $FFFF	0
not -2	1
not -32768	32767

5.4. OPERATORY TYPU MNOŻENIA

Dwuargumentowe operatory * (mnożenia) i / (dzielenia) mogą dotyczyć zarówno danych typu *real*, jak i danych typu *integer*. Dwuargumentowy operator and może dotyczyć zarówno danych typu *boolean*, jak i danych typu *integer*. Pozostałe operatory typu mnożenia mogą dotyczyć jedynie danych typu *integer*. Wymagania te zebrano w tabl. 5.1, w której przytoczono także typ rezultatu operacji.

Sposób wykonywania operacji * i / nie wymaga wyjaśnień. Należy jedynie pamiętać, że rezultatem dzielenia jest zawsze dana typu *real*.

Tabl.5.1. Operatory typu mnożenia

Operator	Argument-1	Argument-2	Rezultat
*	*real*	*real*	*real*
	real	*integer*	*real*
	integer	*real*	*real*
	integer	*integer*	*integer*
/	*real*	*real*	*real*
	real	*integer*	*real*
	integer	*real*	*real*
	integer	*integer*	*real*
div	*integer*	*integer*	*integer*
mod	*integer*	*integer*	*integer*
and	*integer*	*integer*	*integer*
	boolean	*boolean*	*boolean*
shl	*integer*	*integer*	*integer*
shr	*integer*	*integer*	*integer*

Przykłady

Wyrażenie	Rezultat
25 * 40	1000
25 * 40.0	1000.0
0.25 * 40.0	10.0
24 / 3	8.0
24.0 / 3.0	8.0

Rezultatem operacji **div** jest dana typu *integer*, o tak dobranej wartości, że stanowi ona część całkowitą rezultatu podzielenia pierwszego argumentu przez drugi. Operacja **mod** jest natomiast zdefiniowana jako

```
a mod b = a - ((a div b) * b)
```

Przykłady

Wyrażenie	Rezultat
25 div 7	3
-25 div 7	-3
25 mod 7	4
-25 mod 7	-4

Rezultatem operacji **and** na danych typu *boolean* jest koniunkcja tych argumentów wyznaczona jak następuje:

argument a	*argument b*	*a and b*
false	*false*	*false*
true	*false*	*false*
false	*true*	*false*
true	*true*	*true*

Rezultatem operacji **and** na danych typu *integer* jest dana typu *integer*, której reprezentacja stanowi iloczyn logiczny wyznaczony równolegle na wszystkich odpowiadających sobie parach bitów argumentów. Iloczyn logiczny dla pary bitów ma wartość 1 tylko wtedy, kiedy oba bity mają wartość 1. W przeciwnym razie ma on wartość 0.

Przykłady

Wyrażenie	Rezultat
false and *true*	*false*
2 and 2	2
12 and 22	4
-5 and 0	0

Dwuargumentowe operacje **shl** i **shr** mogą dotyczyć jedynie danych typu *integer*. Rezultatem każdej z tych operacji jest również dana typu *integer*. Dana ta ma taką reprezentację, jaka powstaje z reprezentacji pierwszego argumentu po przesunięciu go w lewo (**shl**) albo w prawo (**shr**), o taką liczbę pozycji, jaką określa wartość drugiego argumentu. Przyjmuje się, że przesuwanie ma charakter logiczny. Oznacza to, że podczas przesuwania bit znaku nie ulega powieleniu, a zwolnione pozycje są zapełniane bitami 0.

Przykłady

Wyrażenie	Rezultat
2 shl 1	4
2 shl 3	16
3 shr 1	1
-3 shr 1	32766

■

5.5. OPERATORY TYPU DODAWANIA

Dwuargumentowe operatory + (dodawania) i - (odejmowania) mogą dotyczyć zarówno danych typu *real*, jak i danych typu *integer*. Pozostałe operatory typu dodawania mogą dotyczyć par argumentów identycznych typów: *integer* albo *boolean*. Wymagania te zebrano w tabl. 5.2, w której przytoczono także typ rezultatu operacji.

Tabl.5.2. Operatory typu dodawania

Operator	Argument-1	Argument-2	Rezultat
+	*real*	*real*	*real*
	real	*integer*	*real*
	integer	*real*	*real*
	integer	*integer*	*integer*
-	*real*	*real*	*real*
	real	*integer*	*real*
	integer	*real*	*real*
	integer	*integer*	*integer*
or	*integer*	*integer*	*integer*
	boolean	*boolean*	*boolean*
xor	*integer*	*integer*	*integer*
	boolean	*boolean*	*boolean*

Sposób wykonywania operacji + i - nie wymaga wyjaśnień. Należy jedynie pamiętać, że rezultat operacji jest typu *integer* tylko wtedy, kiedy oba argumenty są tego typu. W przeciwnym razie rezultat operacji jest typu *real*.

Przykłady

Wyrażenie	Rezultat
20 + 40	60
20.0 + 40	60.0
60.0 - 40.0	20.0

■

Rezultatem operacji or na danych typu *boolean* jest alternatywa tych argumentów, wyznaczona jak następuje:

argument *a*	argument *b*	*a* or *b*
false	*false*	*false*
true	*false*	*true*
false	*true*	*true*
true	*true*	*true*

Rezultatem operacji **or** na danych typu *integer* jest dana typu *integer*, której reprezentacja stanowi sumę logiczną wyznaczoną równolegle na wszystkich odpowiadających sobie parach bitów argumentów. Suma logiczna dla pary bitów ma wartość 0 tylko wtedy, gdy oba bity mają wartość 0. W przeciwnym razie ma wartość 1.

Przykłady

Wyrażenie	Rezultat
false **or** *true*	*true*
2 **or** 3	3
12 **or** 22	30
$8000 **or** 1	-32767

Rezultatem operacji **xor** na danych typu *boolean* jest różnica symetryczna tych argumentów, wyznaczona jak następuje:

argument *a*	argument *b*	*a* **xor** *b*
false	*false*	*false*
true	*false*	*true*
false	*true*	*true*
true	*true*	*false*

Rezultatem wykonania operacji **xor** na danych typu *integer* jest dana typu *integer*, której reprezentacja stanowi sumę modulo 2 wyznaczoną równolegle na wszystkich odpowiadających sobie parach bitów argumentów. Suma modulo 2 ma wartość 0 tylko wtedy, gdy oba bity są identyczne. W przeciwnym razie ma wartość 1.

Przykłady

Wyrażenie	Rezultat
true **xor** *true*	*false*
$8000 **xor** 2	-32766
12 **xor** 22	26
-1 **xor** -1	0

5.6. RELACJE

Operatory *relacji* mogą dotyczyć dowolnych danych typu *integer*, *boolean*, *real* i *char*. Rezultatem operacji jest zawsze dana typu *boolean*, o wartości *false* albo *true*. Jeśli pewien argument relacji jest typu *boolean* albo *char*, to drugi musi być takiego samego typu. W przypadku argumentów arytmetycznych nie wymaga się zgodności typów. Umożliwia to np. porównywanie argumentów typu *real* z argumentami typu *integer*.

Operatorami relacji są: = (równe), <> (nie równe), > (większe), < (mniejsze), >= (większe lub równe), <= (mniejsze lub równe).

Przykłady

Wyrażenie	Rezultat
20 = 20	*true*
30.0 > 20	*true*
'J' < 'B'	*false*
false <> *true*	*true*

■

5.7. WYWOŁANIA FUNKCJI

Wywołanie funkcji składa się z identyfikatora funkcji, bezpośrednio po którym występuje ujęta w nawiasy okrągłe lista argumentów wywołania. Wywołanie funkcji bezparametrowej składa się tylko z identyfikatora funkcji.

Przykłady

Eof(Input)
KeyPressed ■

Instrukcja programu stanowi opis czynności, które będą wykonane podczas realizowania algorytmu. W Turbo Pascalu istnieją dwa rodzaje instrukcji: *proste* i *strukturalne*. Instrukcjami prostymi są: instrukcja przypisania, instrukcja wywołania procedury, instrukcja przejścia i instrukcja pusta. Instrukcjami strukturalnymi są: instrukcja grupująca, instrukcja warunkowa, instrukcja wyboru oraz trzy instrukcje iteracyjne.

6.1. INSTRUKCJA PRZYPISANIA

Instrukcja przypisania składa się z następujących po sobie: napisu identyfikującego zmienną, symbolu przypisania i wyrażenia. W obrębie definicji funkcji musi wystąpić co najmniej jedna instrukcja przypisania, w której napisem identyfikującym zmienną jest identyfikator funkcji.

Wykonanie instrukcji przypisania powoduje wyznaczenie danej reprezentowanej przez wyrażenie i przypisanie tej danej zmiennej identyfikowanej przez napis występujący po lewej stronie symbolu przypisania. W przypadku omówionego wyżej przypisania występującego w obrębie definicji funkcji następuje określenie rezultatu funkcji, tj. danej udostępnianej w miejscu wywołania funkcji.

Wymaga się, aby typ wyrażenia oraz typ zmiennej były zgodne w sensie przypisania. Zgodność ta jest zapewniona wtedy, kiedy zmienna i wyrażenie są tego samego typu, jak również wówczas, gdy zmienna jest typu *real*, a wyrażenie typu *integer*. Należy nadmienić, że nie jest dozwolone przypisywanie plików, ale jest dozwolone przypisywanie takich danych strukturalnych jak tablice, rekordy i mnogości.

Składnia

instrukcja-przypisania:
zmienna := wyrażenie

Przykłady

```
i := i + 2
x1 := (-b + sqrt(b * b - 4 * a * c)) / (2 * a)
arr[2,3] := m = n
fun := false
```

■

6.2. INSTRUKCJA WYWOŁANIA PROCEDURY

Instrukcja wywołania procedury składa się z identyfikatora procedury, bezpośrednio po którym następuje ujęta w nawiasy okrągłe lista argumentów wywołania. Instrukcja wywołania procedury bezparametrowej składa się tylko z identyfikatora procedury.

Wykonanie instrukcji wywołania procedury powoduje wykonanie czynności wyszczególnionych w definicji tej procedury. Zanim to nastąpi, odbywa się identyfikacja zmiennych stanowiących argumenty wywołania procedury i wyznaczenie wartości wyrażeń występujących w wywołaniu.

Składnia

instrukcja-wywołania-procedury:
nazwa-procedury (lista-argumentów)
nazwa-procedury
nazwa-procedury:
identyfikator

Przykłady

```
Rewrite(OutFile)
Move(source,target)
Exit
```

■

6.3. INSTRUKCJA PRZEJSCIA

Instrukcja przejścia składa się ze słowa kluczowego **goto**, bezpośrednio po którym następuje odwołanie do etykiety opatrującej instrukcję.

Wykonanie instrukcji przejścia powoduje przejście do wykonywania ciągu instrukcji rozpoczynającego się od instrukcji opatrzonej etykietą. Wymaga to, aby instrukcja przejścia powołująca się na etykietę znajdowała się w zasięgu deklaracji tej etykiety. Zabrania się, aby wykonanie instrukcji

przejścia prowadziło do wnętrza instrukcji strukturalnej albo podprogramu. Zabrania się także, by wykonanie takiej instrukcji powodowało przedwczesne zakończenie wykonywania podprogramu.

Składnia

instrukcja-przejścia:
goto *etykieta*
etykieta:
identyfikator

Przykłady

goto 345
goto *finish* ■

6.4. INSTRUKCJA PUSTA

Instrukcja pusta jest jedyną instrukcją, której zapisanie nie wymaga użycia symboli języka. Występuje ona w tych kontekstach, w których jest wymagane użycie instrukcji, ale chce się uniknąć wykonania jakiejkolwiek akcji.

Wykonanie instrukcji pustej nie wywołuje żadnych skutków.

Składnia

instrukcja-pusta:
puste
puste:

Przykłady (miejsce wystąpienia instrukcji pustej oznaczono komentarzem zawierającym wykrzyknik)

begin { ! } **end**
while *false* **do** { ! };
repeat { ! } **until** *true*
begin { ! } ; { ! } **end**
case *true* **of** *false*: { ! } ; *true*: { ! } **end** ■

6.5. INSTRUKCJA GRUPUJĄCA

*Instrukcja grupująca** składa się ze słowa kluczowego **begin**, bezpośrednio po którym następuje sekwencja instrukcji i słowo kluczowe **end**. Instrukcja grupująca jest używana wszę-

---

*Instrukcja grupująca jest w wielu książkach nazywana instrukcją złożoną, mimo iż jest to jedna z najprostszych instrukcji języka

dzie tam, gdzie składnia języka wymaga użycia jednej instrukcji, a niezbędne jest wykonanie sekwencji instrukcji.

Wykonanie instrukcji grupującej powoduje wykonanie zawartej w niej sekwencji instrukcji.

Składnia

instrukcja-grupująca:
begin *sekwencja-instrukcji* **end**

Przykłady

```
begin
  i := 2; j := 2
end

begin
 Writeln(OutFile);
 Close(OutFile)
end
```

■

6.6. INSTRUKCJA WARUNKOWA

Instrukcja warunkowa składa się ze słowa kluczowego **if**, bezpośrednio po którym następuje wyrażenie logiczne, słowo kluczowe **then** i instrukcja. Po instrukcji może występować słowo kluczowe **else** i ponownie instrukcja.

Wykonanie instrukcji warunkowej składa się z wyznaczenia wartości wyrażenia, a następnie, jeśli rezultat jest daną o wartości *true*, wykonania instrukcji następującej po słowie kluczowym **then**. Jeśli rezultat jest daną o wartości *false*, a instrukcja warunkowa nie zawiera słowa kluczowego **else**, to jej wykonanie uznaje się za zakończone. Jeśli zawiera takie słowo, to jest wykonywana instrukcja występująca po tym słowie.

Jeśli instrukcja zawarta w instrukcji warunkowej jest także instrukcją warunkową, to może ona wystąpić przed słowem kluczowym **else** ale tylko wtedy, gdy także zawiera to słowo.

Składnia

instrukcja-warunkowa:
if *wyrażenie-logiczne* **then** *instrukcja*
if *wyrażenie-logiczne* **then** *instrukcja*
else *instrukcja*

Przykłady

```
if a = b then a := 5 else b := 5

if a = b then begin
```

```
    a := 6;
    b := 6
end

if a = b then
  if a = 5 then
    b := 2
  else
    a := 3
else
  a := 13
```

■

6.7. INSTRUKCJA WYBORU

Instrukcja wyboru składa się ze słowa kluczowego **case**, bezpośrednio po którym występuje wyrażenie selekcyjne, słowo kluczowe **of**, sekwencja instrukcji wyboru poprzedzonych etykietami wyboru, a po niej słowo kluczowe **end**.

Etykieta wyboru składa się z listy literałów, po której następuje symbol : (dwukropek). Każdy z literałów tej listy musi być tego samego typu jak wyrażenie selekcyjne. Jeśli pewna podlista listy literałów stanowi ciąg kolejnych elementów typu porządkowego, to może być zastąpiona konstrukcją

pierwszy..ostatni

wyszczególniającą pierwszy i ostatni element tego ciągu. Zamiast etykiet wyboru poprzedzających ostatnią instrukcję sekwencji instrukcji wyboru może być użyte słowo kluczowe **else**. Po takiej "etykiecie" nie stawia się dwukropka.

Składnia

instrukcja-wyboru:
case *wyrażenie-selekcyjne* **of**
sekwencja-instrukcji-wyboru **end**

Wykonanie instrukcji wyboru składa się z wyznaczenia wartości wyrażenia selekcyjnego, a następnie wykonania tej instrukcji należącej do sekwencji instrukcji wyboru, która jest poprze- dzona etykietą wyboru o wartości równej wartości wyrażenia selekcyjnego. Jeśli takiej instrukcji nie ma, to jest wykony- wana instrukcja poprzedzona "etykietą" wyboru ze słowem kluczowym **else**, a jeśli i takiej nie ma, to jest wykonywana instrukcja pusta.

Przykłady

```
case a = b of
  false: i := 5;
```

```
    true:   j := 2
  end

  case Operator of
    '+': Result := Result + 1;
    '-': Result := Result - 1;
    else goto fin
  end

  case Year of
    1945..1955 : Writeln('hard work');
    1956..1969 : Writeln('hopes');
    1970..1979 : Writeln('prosperity');
    1980..1981 : Writeln('euphoria');
    else         Writeln('no comment')
  end                                          ■
```

6.8. INSTRUKCJE ITERACYJNE

Instrukcje iteracyjne służą do organizowania cykli programowych. Występują one w trzech odmianach, jako tzw. instrukcje for, while i repeat.

Instrukcja for

Instrukcja for składa się ze słowa kluczowego **for**, bezpośrednio po którym następuje identyfikator zmiennej sterującej cyklu, symbol przypisania, wyrażenie określające wartość początkową zmiennej sterującej, słowo kluczowe **to** albo **downto**, wyrażenie określające wartość końcową zmiennej sterującej, słowo kluczowe **do** oraz dowolna instrukcja.

Zmienna sterująca cyklu oraz oba wyrażenia muszą być tego samego typu porządkowego.

Składnia

instrukcja-for:

for *zmienna* := *wyrażenie*$_a$ **to** *wyrażenie*$_b$ **do** *instrukcja*

for *zmienna* := *wyrażenie*$_a$ **downto** *wyrażenie*$_b$ **do** *instrukcja*

Wykonanie instrukcji for powoduje wykonywanie zawartej w niej instrukcji dla tych wszystkich wartości zmiennej sterującej, które są zawarte w przedziale domkniętym wyznaczonym przez wartości *wyrażenia*$_a$ i *wyrażenia*$_b$.

Jeśli w instrukcji for wystąpiło słowo kluczowe **to**, to zmienna sterująca przybiera wartości od najmniejszej do największej. Jeśli natomiast wystąpiło słowo kluczowe **downto**, to

przybiera ona wartości od największej do najmniejszej. Jeśli w pierwszym z tych dwóch przypadków prawdziwa jest relacja

wyrażenie$_a$ > *wyrażenie*$_b$

albo gdy w drugim jest prawdziwa relacja

wyrażenie$_a$ < *wyrażenie*$_b$

to cykl wyznaczony przez instrukcję for nie jest wykonywany w ogóle. W takim przypadku wykonanie instrukcji for ogranicza się do wyznaczenia wartości obu wymienionych wyrażeń.

W pozostałych przypadkach cykl jest wykonywany co najmniej jednokrotnie, a po jego zakończeniu zmienna sterująca cyklu ma wartość *wyrażenia*$_b$.

Pozostaje nadmienić, że wyznaczenie wartości omawianych wyrażeń jest jednokrotne, a jawne przypisywanie danych zmiennej sterującej cyklu jest zabronione.

Przykłady

```
for i := 2 to 8 do Writeln(i)

for j := 8 downto 2 do Writeln(j)

for toggle := false to true do
  for letter := 'i' to 'n' do
     arr[toggle,letter] := 2
```

■

Instrukcja while

Instrukcja while składa się ze słowa kluczowego while, bezpośrednio po którym następuje wyrażenie typu *boolean*, słowo kluczowe do oraz dowolna instrukcja.

Składnia

instrukcja-while:
while *wyrażenie* **do** *instrukcja*

Wykonanie instrukcji while przebiega według poniższego algorytmu:

1. Wyznaczana jest wartość wyrażenia.
2. Jeśli wartością tą jest *false*, to wykonanie instrukcji uznaje się za zakończone.
3. Jeśli wartością tą jest *true*, to jest wykonywana instrukcja następująca po słowie kluczowym do, a po tym następuje powtórzenie opisanych czynności od początku.

Przykłady

```
while i <> 0 do begin
   i := i - 1;
   Writeln(i)
end
```

```
while Flag do Fun(Flag)                                    ■
```

Instrukcja repeat

Instrukcja repeat składa się ze słowa kluczowego **repeat**, bezpośrednio po którym następują: dowolna instrukcja, słowo kluczowe **until** i wyrażenie typu *boolean*.

Składnia

instrukcja-repeat:
repeat *instrukcje* **until** *wyrażenie*

Wykonanie instrukcji repeat przebiega według poniższego algorytmu:

1. Wykonywane są instrukcje następujące po słowie kluczowym **repeat**.
2. Wyznaczana jest wartość wyrażenia występującego po słowie kluczowym **until**.
3. Jeśli wartością tą jest *true*, to wykonanie instrukcji uznaje się za zakończone.
4. Jeśli wartością tą jest *false*, to następuje powtórzenie opisanych czynności od początku.

Przykłady

```
repeat
 Writeln(i);
 i := i - 1
until i = 0

repeat until KeyPressed                                    ■
```

Podstawowymi *typami danych* w Turbo Pascalu są *typy skalarne*, a wśród nich *typy porządkowe*. Każdy z typów porządkowych ma tę właściwość, że definiujący go zbiór jest przeliczalny. Należy odnotować, że typ *real*, mimo iż jest typem skalarnym, nie jest typem porządkowym. To właśnie jest przyczyną, dla której wyrażenie selekcyjne instrukcji wyboru, etykiety instrukcji wyboru oraz zmienne sterujące i wyrażenia występujące w instrukcji for nie mogą być typu *real*.

7.1. TYPY WYLICZENIOWE

Typy wyliczeniowe stanowią ważną odmianę typów porządkowych. Pozostają one w związku ze zbiorami o niewielkiej liczbie elementów, na których nie wykonuje się operacji arytmetycznych. Poszczególne elementy tych zbiorów są oznaczane unikalnymi identyfikatorami, a ich uporządkowanie wynika z kolejności wystąpienia tych identyfikatorów w opisie typu wyliczeniowego.

Opis typu wyliczeniowego składa się z listy identyfikatorów elementów tego typu, ujętej w nawiasy okrągłe. Każdy z takich identyfikatorów pełni rolę literału danego typu. Kolejność wystąpienia identyfikatorów na liście jest istotna.

Składnia

opis-typu-wyliczeniowego:
(*lista-identyfikatorów*)

Przykłady

```
type
  Day = (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
```

```
Color = (Heart,Diamond,Spade,Club);
RGB = (Red,Green,Blue);
```

● Dane typu *Day* są identyfikowane przez trzyliterowe nazwy symboliczne oznaczające dni tygodnia.

● Dane typu *Color* są identyfikowane przez nazwy symboliczne *Heart*, *Diamond*, *Spade* i *Club*.

● Dane typu *RGB* są identyfikowane przez nazwy symboliczne *Red*, *Green* i *Blue*. ■

W zasięgu opisu typu wyliczeniowego nie jest dozwolone użycie innego opisu typu wyliczeniowego, powołującego się na literały występujące w pierwszym z tych opisów. Ograniczenie to nie dotyczy jednak predefiniowanego typu wyliczeniowego *boolean*, skąd wynika, że jest dozwolone posłużenie się np. opisem typu (*false*,*true*).

Przykład

```
type
  Day = (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
  WeekEnd = (Sat,Sun);
```

● Przytoczony zestaw definicji typu jest niepoprawny, ponieważ przecięcie list literałów typów *Day* i *WeekEnd* nie jest puste. ■

Ponieważ typy wyliczeniowe są związane ze zbiorami przeliczalnymi, każdemu elementowi takiego zbioru można przypisać liczbę nieujemną, określającą miejsce elementu na liście identyfikatorów typu wyliczeniowego. W tym sensie dla definicji

```
type
  Direction = (North,South,East,West)
```

można przypisać identyfikatorowi *North* liczbę 0, identyfikatorowi *South* liczbę 1, identyfikatorowi *East* liczbę 2 i identyfikatorowi *West* liczbę 3. Dane o wartości tych liczb można uzyskać posługując się predefiniowaną funkcją *ord*. Argumentem tej funkcji jest wyrażenie reprezentujące daną typu wyliczeniowego, a rezultatem - nieujemna dana typu *integer*, której wartość określa liczony od 0 numer pozycji danej w zawierającym ją zbiorze uporządkowanym.

Dwiema innymi funkcjami, które mogą dotyczyć danych typu wyliczeniowego, są *pred* i *succ*. Argumentem funkcji *pred* może być dowolne wyrażenie typu wyliczeniowego, reprezentujące daną zbioru uporządkowanego. Rezultatem tej funkcji jest ta dana tego zbioru, która bezpośrednio poprzedza daną reprezentowaną przez argument. Zabrania się, aby argument funkcji

pred reprezentował pierwszą daną zbioru, jako że ten element nie ma poprzednika.

Analogiczną do *pred* jest funkcja *succ*. Jej argumentem może być także dowolne wyrażenie typu wyliczeniowego, reprezentujące daną zbioru uporządkowanego. Rezultatem funkcji *succ* jest ta dana tego zbioru, która bepośrednio następuje po danej reprezentowanej przez argument. Zabrania się, aby argument funkcji *succ* reprezentował ostatnią daną zbioru, jako że ten element nie ma następnika.

Argumentami funkcji *ord*, *pred* i *succ* mogą być także wyrażenia reprezentujące dane typu *integer* i *byte*. W takim przypadku przyjmuje się, że jeśli użycie funkcji jest poprawne, to obowiązują tożsamości

ord(*n*) = *n*
pred(*n*) = *n* - 1
succ(*n*) = *n* + 1

Przykłady. W zasięgu definicji predefiniowanego typu *boolean* oraz zdefiniowanego jawnie typu *Color*

```
type
  Color = (Heart,Diamond,Spade,Club);
```

prawdziwe jest następujące zestawienie

Wyrażenie	Rezultat
ord(*false*)	0
pred(*true*)	*false*
ord(*Club*)	3
succ(*Diamond*)	*Spade*

■

7.2. TYPY OKROJONE

Typy okrojone stanowią drugą po typach wyliczeniowych odmianę typów porządkowych. Pozostają one w związku ze zbiorami tych elementów wybranego zbioru uporządkowanego, których liczby porządkowe uzyskane za pomocą funkcji *ord* tworzą ciąg arytmetyczny o różnicy 1. Typ okrojony zachowuje wszystkie właściwości swego typu macierzystego, z tym tylko wyjątkiem, że dane typu okrojonego muszą należeć do podzbioru stanowiącego okrojenie typu macierzystego.

Opis typu okrojonego składa się z literału stanowiącego ograniczenie dolne zakresu typu okrojonego, symbolu .. (para kropek) oraz literału stanowiącego ograniczenie górne zakresu. Wymaga się, aby zakres wyznaczony przez wymienione tu literały nie był pusty.

Składnia

opis-typu-okrojonego:
ograniczenie-dolne .. ograniczenie-górne
ograniczenie:
literał
nazwa-literału

Przykłady

```
type
  Quadrant = 0..90;
  Upper = 'A'..'Z';
  Logical = false..true;
  Colour = (Red,Blue,Green,Yellow,Orange);
  Hue = Blue..Yellow;
```

● Elementy zbioru danych typu *Quadrant* są reprezentowane przez liczby typu *integer* z przedziału od 0 do 90 włącznie.

● Elementy zbioru danych typu *Upper* są reprezentowane przez literały typu *char* z przedziału od *'A'* do *'Z'*.

● Elementy zbioru danych typu *Logical* są reprezentowane przez literały *false* i *true*, oba typu *boolean*.

● Elementy zbioru danych typu *Hue* są reprezentowane przez literały wyliczeniowe *Blue*, *Green* i *Yellow*, wszystkie typu *Colour*. ■

Pozostaje nadmienić, że predefiniowany typ *byte* jest (z pewnymi ograniczeniami) typem okrojonym typu *integer*, tj. że obowiązuje dla niego niejawna definicja

```
type
  byte = 0..255;
```

W odniesieniu do danych typów okrojonych mogą być także stosowane funkcje *ord*, *pred* i *succ*. Wyznaczanie rezultatów tych funkcji odbywa się wówczas w typie macierzystym.

Przykłady. W zasięgu definicji typu *Hue*

```
type
  Colour = (Red,Blue,Green,Yellow,Orange);
  Hue = Blue..Yellow;
```

prawdziwe jest następujące zestawienie:

Wyrażenie	Rezultat
ord(Blue)	1
succ(Yellow)	*Orange*
pred(Yellow)	*Green*

■

Należy zdecydowanie podkreślić, że posługiwanie się typami wyliczeniowymi i okrojonymi zwiększa czytelność progra-

mów, a ponadto ułatwia ich uruchamianie, jako że umożliwia automatyczne wykonywanie badań poprawności zakresu wartości danych. Nie bez znaczenia jest także fakt, że posługiwanie się danymi wymienionych typów prowadzi do oszczędności pamięci. Wynika to stąd, że jeśli liczba elementów zbioru macierzystego związanego z typem danej nie przekracza 255, to Turbo Pascal reprezentuje daną w zaledwie jednym bajcie pamięci. Równie oszczędna reprezentacja jest przyjmowana także w odniesieniu do typów okrojonych z typem macierzystym *integer*, jeśli oba ograniczenia zakresu mieszczą się w przedziale domkniętym [0,255].

7.3. KONWERSJE TYPOW

W języku Turbo Pascal rozwinięto koncepcję funkcji *ord*, dokonującej konwersji danej dowolnego typu porządkowego w daną typu *integer*. Wprowadzono mianowicie rodzinę operatorów jednoparametrowych, o nazwach identycznych z nazwami typów porządkowych, umożliwiających wykonywanie konwersji danych między dowolnymi typami porządkowymi.

Operator konwersji ma postać *opr(arg)*, gdzie *opr* jest identyfikatorem typu porządkowego, *arg* zaś jest dowolnym wyrażeniem porządkowym. Rezultatem takiej operacji jest dana typu *opr*, tak dobrana, że prawdziwa jest relacja *ord(a)* = = *ord(opr(a))*.

Przykłady. W zasięgu definicji predefiniowanych typów porządkowych oraz zdefiniowanego jawnie typu *Color*

```
type
  Color = (Heart,Diamond,Spade,Club);
```

prawdziwe jest następujące zestawienie:

Wyrażenie	Rezultat
integer(Club)	3
Color(2)	*Spade*
char(65)	*'A'*
integer('A')	65
boolean(Heart)	*false*
byte(true)	1

■

7.4. KONTROLA POPRAWNOSCI ZAKRESU

Jedną z właściwości kompilatora języka Turbo Pascal jest możliwość takiego skompilowania programu źródłowego, aby

podczas jego wykonywania były przeprowadzane kontrole poprawności zakresu danych skalarnych. Ponieważ kontrole takie spowalniają wykonywanie programu, są wstępnie wyłączone i mogą być włączane za pomocą dyrektywy {$R+}. W tych zakresach programu źródłowego, w których kontrole są zbędne, można je wyłączyć, posługując się dyrektywą {$R-} .

Przykład. Włączanie i wyłączanie kontroli zakresów danych

```
program Check;
type
  Figure = (Square,Circle,Diamond,Triangle);
var
  Shape : Figure;
begin
  Shape := Circle;
  Shape := Figure(4);
  {$R+}
  Shape := succ(Triangle);
  {$R-}
  Shape := pred(Square)
end.
```

● Kontrola poprawności zakresu jest włączona jedynie na czas wykonywania instrukcji przypisania zawartej między dyrektywami kompilacji.

● Gdyby z programu usunąć dyrektywy kompilacji, to zostałby on wykonany bez sygnalizowania błędów.

● Wykonanie programu w podanej postaci spowoduje zasygnalizowanie błędu przekroczenia zakresu danych. Stanie się to podczas wyznaczania rezultatu funkcji succ. ■

Każdy z typów łańcuchowych jest typem złożonym. Podczas definiowania typu łańcuchowego jest określana maksymalna liczba znaków, z których może składać się dana łańcuchowa tego typu. Z każdym typem łańcuchowym jest związany zbiór ciągów znaków. Należy do niego pusty ciąg znaków oraz ciągi o długości nie przekraczającej maksymalnej liczby znaków.

Opis typu łańcuchowego składa się ze słowa kluczowego **string**, bezpośrednio po którym następuje zawarta w nawiasach kwadratowych maksymalna liczba znaków łańcucha danego typu. Liczba ta może być wyrażona za pomocą literału typu *integer* albo za pomocą równoważnej takiemu literałowi nazwy literału.

Składnia

opis-typu-łańcuchowego:
string [*rozmiar*]
rozmiar:
literał
nazwa-literału

Przykłady

```
type
  Cities = string[20];
  Names = string[12];
```

■

Każda z danych typu łańcuchowego zajmuje o jeden więcej bajtów pamięci niż wynosi maksymalna liczba znaków łańcucha danego typu. W tym dodatkowym bajcie jest przechowywany bieżący rozmiar łańcucha. Z tej informacji wynika, że maksymalny rozmiar łańcucha nie może przekroczyć 255 znaków.

8.1. ZŁĄCZENIA ŁAŃCUCHÓW

Złączenie pary łańcuchów w jeden łańcuch może być zrealizowane za pomocą *operatora konkatenacji*. Operator ten jest zapisywany za pomocą znaku + (plus) i może dotyczyć pary wyrażeń łańcuchowych dowolnych typów. Złączenie pary łańcuchów jest łańcuchem składającym się z wszystkich znaków pierwszego łańcucha i następujących po nich wszystkich znaków drugiego łańcucha. Wymaga się, aby rozmiar łańcucha, który jest rezultatem złączenia, nie przekroczył 255 znaków.

Przykłady

Wyrażenie	Rezultat
'jan' + 'ewa'	*'janewa'*
'5' + '.' + '4'	*'5.4'*
'j' + '' + 'b'	*'jb'*

(w ostatnim przykładzie wystąpił literał łańcuchowy reprezentujący pusty ciąg znaków) ■

8.2. RELACJE

Dowolne pary łańcuchów mogą być porównywane za pomocą *operatorów relacji*. Operatorami relacji są: = (równe), <> (nie równe), < (mniejsze), > (większe), <= (mniejsze lub równe), >= (większe lub równe). Priorytet każdego z operatorów relacji jest niższy niż priorytet operatora złączenia. Umożliwia to konstruowanie wyrażeń takich jak np. *'jan' = 'j' + 'an'* bez konieczności posługiwania się nawiasami.

Dwa łańcuchy są równe tylko wówczas, gdy są takiej samej długości oraz gdy składają się z identycznych znaków. Jeśli nie są równe, to relacja dotycząca łańcuchów zostaje zastąpiona relacją pierwszej pary odpowiadających sobie, ale różnych znaków. Jeśli para taka nie zostanie znaleziona, ponieważ początkowym fragmentem jednego z łańcuchów jest drugi łańcuch, to za mniejszy zostanie uznany ten łańcuch, który jest krótszy.

Przykłady

Wyrażenie	Rezultat
'A' > 'B'	*false*
'jan' > 'Jan'	*true*
'' < char(0)	*true*
'' = ''	*true*

```
'Jan' <= 'Jane'        true
'2599' < '270'         true                    ■
```

8.3. PRZYPISANIA

Przypisania łańcuchów zmiennym łańcuchowym mogą być realizowane za pomocą instrukcji przypisania. Z prawej strony symbolu przypisania może występować dowolne wyrażenie łańcuchowe, z lewej natomiast - nazwa zmiennej łańcuchowej. Jeśli długość łańcucha reprezentowanego przez wyrażenie przekracza maksymalny rozmiar łańcucha, który może być reprezentowany w zmiennej, to przypisanie odbywa się od lewej do prawej z pominięciem znaków nadmiarowych.

Przykład

```
type
  Name : string[7];
var
  FirstName,LastName : Name;
...
  FirstName := 'Jan';
  LastName := 'Bielecki';
```

- Zmiennej *FirstName* zostanie przypisana dana łańcuchowa o wartości literału *'Jan'*.
- Zmiennej łańcuchowej *LastName* zostanie przypisana dana łańcuchowa o wartości literału *'Bieleck'*. ■

8.4. PODPROGRAMY

Rozszerzeniem zbioru operacji łańcuchowych są *funkcje* i *podprogramy łańcuchowe*. Ich użycie znakomicie ułatwia przetwarzanie danych łańcuchowych.

Funkcja Length

Wywołanie: *Length(Str)*

Przyjmuje się, że *Str* jest wyrażeniem łańcuchowym reprezentującym ciąg znaków.

Rezultatem funkcji *Length* jest dana typu *integer* o wartości określającej liczbę znaków tego ciągu.

Przykłady

Wyrażenie	Rezultat
Length('jb')	2
Length('''')	1
Length('')	0

■

Procedura Str

Wywołanie: *Str(Val,Var)*

Przyjmuje się, że *Val* jest wyrażeniem arytmetycznym, a *Var* jest nazwą zmiennej łańcuchowej.

Wykonanie procedury *Str* powoduje przekształcenie danej reprezentowanej przez wyrażenie *Val* w ciąg znakowy mający postać literału o wartości tej danej. Następnie ciąg ten zostanie przekształcony w daną łańcuchową i przypisany zmiennej *Var*.

Jeśli wyrażenie *Val* jest typu *integer*, to przyjmuje się, że literał ma postać liczby całkowitej o minimalnej liczbie cyfr.

Jeśli bezpośrednio po wyrażeniu *Val* występuje kwalifikator o postaci :*m*, w którym *m* jest wyrażeniem typu *integer*, to określa on długość ciągu znaków, na który zostanie przekształcona dana reprezentowana przez *Val*. W ciągu takim wymieniony wyżej literał jest wyrównany prawostronnie. Jeśli literał nie da się przedstawić za pomocą *m* znaków, to *m* zostanie niejawnie zwiększone o tyle, aby było to wykonalne.

Przykłady (przyjęto, że *StrVar* jest typu **string**[4])

Wywołanie	Ciąg	*StrVar*
Str(45,StrVar)	45	'45'
Str(-250,StrVar)	-250	'-250'
Str(45:3,StrVar)	b45	' 45'
Str(45:5,StrVar)	bbb45	' 4'
Str(-2:5,StrVar)	bbb-2	' -'

■

Jeśli wyrażenie *Val* jest typu *real*, to przyjmuje się, że ciąg wzorcowy zawierający literał ma postać

bsd.ddddddddddEsdd

w której *b* jest spacją, *s* jest znakiem mantysy lub wykładnika, a *d* jest cyfrą. Znak mantysy jest reprezentowany za pomocą spacji lub znaku - (minus), a znak wykładnika jest reprezentowany za pomocą znaku + (plus) albo - (minus).

Jeśli bezpośrednio po wyrażeniu *Val* występuje kwalifikator o postaci :*m*, w którym *m* jest wyrażeniem typu *integer*,

to określa on długość ciągu znaków, na który zostanie przekształcona dana reprezentowana przez *Val*. W ciągu takim wymieniony wyżej literał zostanie wyrównany prawostronnie. Jeśli literał nie da się przedstawić za pomocą *m* znaków, to z wymienionego wyżej ciągu wzorcowego zostaną w pierwszej kolejności usunięte spacje wiodące, a jeśli i to nie poskutkuje, to zostanie odpowiednio zmniejszona liczba cyfr mantysy, ale nie mniej niż do 2 cyfr. W przypadku redukowania liczby cyfr mantysy będzie stosowane zaokrąglanie.

Przykłady (przyjęto, że *Exp* ma wartość 45.678E1, *StrVar* zaś jest typu **string**[10])

Wywołanie	Ciąg	*StrVar*
Str(*Exp*:9,*StrVar*)	4.568E+02	'4.568E+02'
Str(*Exp*:8,*StrVar*)	4.57E+02	'4.57E+02'
Str(*Exp*:7,*StrVar*)	4.6E+02	'4.6E+02'
Str(*Exp*:6,*StrVar*)	4.6E+02	'4.6E+02'
Str(*Exp*:10,*StrVar*)	4.5678E+02	'4.5678E+02'
Str(*Exp*:11,*StrVar*)	4.5678E+02	'4.5678E+0' ■

Jeśli bezpośrednio po *Val* występuje kwalifikator o postaci :*m*:*n*, w którym *m* i *n* są wyrażeniami typu *integer*, to omawiany literał jest wyrównywany prawostronnie w ciągu o długości *m* i niezależnie od wartości *m* jest przedstawiany jako zaokrąglony literał stałopozycyjny z *n* cyframi ułamkowymi. Jeśli literał nie da się przedstawić za pomocą *m* znaków, to *m* zostanie niejawnie zwiększone o tyle, aby było to wykonalne.

Przykłady (przyjęto, że *Exp* ma wartość 45.678E1, *StrVar* zaś jest typu **string**[10])

Wywołanie	Ciąg	*StrVar*
Str(*Exp*:6:2,*StrVar*)	456.78	'456.78'
Str(-*Exp*:6:2,*StrVar*)	-456.78	'-456.78'
Str(-*Exp*:8:2,*StrVar*)	b-456.78	' -456.78'
Str(-*Exp*:12:2,*StrVar*)	bbbbb-456.78	' -456.78'
Str(-*Exp*:0:2,*StrVar*)	-456.78	'-456.78' ■

Procedura Val

Wywołanie: *Val*(*Str*,*Var*,*Rep*)

Przyjmuje się, że *Str* jest wyrażeniem łańcuchowym, *Var* jest nazwą zmiennej typu *integer* albo *real*, a *Rep* jest nazwą zmiennej typu *integer*.

Wykonanie procedury *Val* powoduje potraktowanie znaków danej reprezentowanej przez *Str* jako zapisu liczby i przypisanie danej o wartości tej liczby zmiennej *Var*. Zabrania się, aby wspomniana liczba była poprzedzona spacjami, jak również zabrania się, aby po niej następowały spacje. Jeśli *Var* jest nazwą zmiennej typu *integer*, to wymaga się, aby także i liczba była typu *integer*.

Jeśli opisana konwersja jest wykonalna, to zmiennej *Rep* zostanie przypisana dana o wartości 0. W przeciwnym razie, zmiennej tej zostanie przypisana dana o wartości dodatniej. Wartość tej danej będzie stanowić numer tego znaku danej łańcuchowej, który uniemożliwił dokonanie konwersji. W takim przypadku wartość zmiennej *Var* nie ulegnie zmianie.

Przykłady (przyjęto, że *IntVar* i *IntRep* są nazwami zmiennych typu *integer*)

Wywołanie	*IntVar*	*IntRep*
Val('23',IntVar,IntRep)	23	0
Val('23.0',IntVar,IntRep)	b.z.	3
Val('-200 ',IntVar,IntRep)	b.z.	5

■

Funkcja Copy

Wywołanie: *Copy(Str,Pos,Num)*

Przyjmuje się, że *Str* jest wyrażeniem łańcuchowym, a *Pos* i *Num* są wyrażeniami typu *integer*.

Rezultatem funkcji *Copy* jest dana łańcuchowa składająca się z *Num* znaków łańcucha reprezentowanego przez *Str*, począwszy od znaku o numerze *Pos*. Jeśli *Pos>Length(Str)*, to rezultatem funkcji jest dana łańcuchowa reprezentująca pusty ciąg znaków. Jeśli *(Pos+Num)>Length(Str)*, to przyjmuje się *Num:=Length(Str)-Pos+1*. Zabrania się, aby *Pos* wykraczało poza zakres 1..255.

Przykłady

Wywołanie	Rezultat
Copy('Branka',3,4)	*'anka'*
Copy('Branka',5,3)	*'ka'*
Copy('Branka',7,2)	*''*
Copy('Branka',2,1)	*'r'*

■

Funkcja Concat

Wywołanie: *Concat(St1,St2, ... ,Stk)*

Przyjmuje się, że *St1*, *St2*, ... ,*Stk* są wyrażeniami

łańcuchowymi.

Rezultatem funkcji jest dana łańcuchowa *St1+St2+...+Stk*. Zabrania się, aby suma długości argumentów przekraczała 255.

Przykłady

Wywołanie	Rezultat
Concat('j','a','n')	*'jan'*
Concat('j','','b')	*'jb'*
Concat('jan',' ','b')	*'jan b'* ■

Funkcja Pos

Wywołanie: *Pos(Src,Trg)*

Przyjmuje się, że *Src* i *Trg* są wyrażeniami łańcuchowymi.

Rezultatem funkcji *Pos* jest dana typu *integer*, której wartość określa najmniejszy numer tej pozycji w ciągu znaków reprezentowanym przez *Trg*, od której rozpoczyna się podciąg identyczny z podciągiem reprezentowanym przez *Src*. Jeśli podciąg taki nie występuje, to rezultatem funkcji jest dana o wartości 0.

Przykłady

Wywołanie	Rezultat
Pos('23','123123')	2
Pos('21','123123')	0
Pos('c','abcabc')	3
Pos('','izabela')	0 ■

Procedura Insert

Wywołanie: *Insert(Src,Trg,Pos)*

Przyjmuje się, że *Src* jest wyrażeniem łańcuchowym, *Trg* jest nazwą zmiennej łańcuchowej, a *Pos* jest wyrażeniem całkowitym.

Wykonanie procedury *Insert* powoduje wstawienie do ciągu znaków reprezentowanego przez *Trg*, w miejscu przed jego znakiem określonym przez *Pos*, ciągu znaków reprezentowanego przez *Src*. Jeśli *Pos>Length(Trg)*, to przyjmuje się, że *Pos:=Length(Trg)+1*. Zabrania się, aby *Pos* wykraczało poza zakres 1..255.

Jeśli wstawienie znaków spowodowałoby przekroczenie maksymalnego rozmiaru zmiennej *Trg*, to przypisanie jej danej reprezentującej nowy ciąg znaków odbędzie się od lewej do prawej z odrzuceniem znaków nadmiarowych.

Przykłady (przyjęto, że zmienna *Trg* jest typu **string**[5], a przed każdym wykonaniem procedury *Insert* ma wartość *'jb'*)

Wywołanie	Trg
Insert('an',Trg,2)	*'janb'*
Insert('jb',Trg,4)	*'jbjb'*
Insert('Janek',Trg,1)	*'janek'*

■

Procedura Delete

Wywołanie: *Delete(Str,Pos,Num)*

Przyjmuje się, że *Str* jest nazwą zmiennej łańcuchowej, a *Pos* i *Num* są wyrażeniami całkowitymi.

Wykonanie procedury *Delete* powoduje usunięcie z ciągu znaków reprezentowanego przez *Str* porcji *Num* znaków, począwszy od pozycji *Pos*. Jeśli *Pos>Length(Str)*, to wykonanie procedury nie ma żadnych skutków. Jeśli *Pos+Num-1>Length(Str)*, to przyjmuje się, że *Num:=Pos+Num-1*. Zabrania się, aby *Pos* wykraczało poza zakres 1..255.

Przykłady (przyjęto, że zmienna *Trg* jest typu **string**[5], a przed każdym wykonaniem procedury ma wartość *'janek'*)

Wywołanie	Trg
Delete(Trg,4,2)	*'jan'*
Delete(Trg,6,1)	*'janek'*
Delete(Trg,5,3)	*'jane'*

■

8.5. TYPY ŁAŃCUCHOWE I ZNAKOWE

Wszystkie typy łańcuchowe oraz predefiniowany typ porządkowy *char* są ze sobą zgodne. Oznacza to, że w tych wszystkich miejscach programu, w których poprawne jest użycie wyrażenia łańcuchowego, poprawne jest odwołanie do danej typu *char* i na odwrót. Wymaga się jedynie, aby podczas przypisywania wyrażenia łańcuchowego zmiennej typu *char* dana reprezentowana przez to wyrażenie była ciągiem znakowym o długości 1.

Dostęp do poszczególnych znaków zmiennej łańcuchowej można uzyskać nie tylko za pomocą predefiniowanej funkcji *Copy*, ale znacznie łatwiej za pomocą indeksowania. Jako definicję indeksowania można przyjąć równoważność

St[i] = Copy(St,i,1)

Równoważność ta jest jednak prawdziwa tylko wtedy, gdy zachodzi relacja *1<=i<=Length(St)*. Poza tym zakresem możliwe

jest sięganie do sąsiednich bajtów pamięci operacyjnej, a w szczególności posługiwanie się wyrażeniem *ord(St[0])* w celu określenia bieżącej długości ciągu znakowego reprezentowanego przez *St*.

Należy nadmienić, że użycie dyrektywy kompilatora {$*R*+} tylko w niewielkim stopniu umożliwia wykrycie błędów związanych z niepoprawnym indeksowaniem. Wynika to stąd, że dyrektywa ta włącza kontrolę przekroczenia maksymalnego dopuszczalnego indeksu, znanego z deklaracji, ale nie wprowadza kontroli odwoływania się poza zakres bieżący.

Przykład

```
var
  Name : string[5];
  ...
Name := 'Jan';
{$R+}
Name[0] := chr(5);
Name[4] := 'e';
Name[5] := 'k';
```

● Po wykonaniu przytoczonych przypisań zmienna *Name* ma wartość *'Janek'*.

● Użycie dyrektywy kompilatora {$*R*+} jest zbędne, ponieważ nie ma wpływu na przebieg wykonywania przypisań.

● Gdyby w zasięgu wspomnianej dyrektywy wystąpiło np. przypisanie

```
Name[6] := 'B'
```

to wykonywanie programu zostałoby wstrzymane na skutek przekroczenia maksymalnego dopuszczalnego zakresu indeksu. ■

Każdy z typów tablicowych jest typem złożonym. *Tablica* składa się z ustalonej liczby komponentów, z których każdy jest takiego samego typu. Komponenty mogą być zarówno typu prostego, jak i typu złożonego. Dostęp do komponentów tablicy, nazywanych również *elementami*, odbywa się poprzez *indeksowanie*. *Indeksem* może być dowolne wyrażenie skalarne ujęte w nawiasy kwadratowe. Dopuszczalny zakres indeksów jest określony w opisie typu tablicowego.

Opis typu tablicowego składa się ze słowa kluczowego array, bezpośrednio po którym następuje ujęty w nawiasy kwadratowe opis indeksów, słowo kluczowe of, a za nim opis typu komponentów.

Składnia

opis-typu-tablicowego:
    array [*typ-indeksów*] of *typ-elementów*
typ-indeksów:
    opis-typu-porządkowego
typ-elementów:
    opis-typu

Przykłady

```
type
  Color = (Red,Green,Blue);
  TruthTable = array[boolean] of boolean;
  Height = 0..200;
var
  Square : array [Color] of Height;
  Matrix : array[2..8] of array[2..8] of integer;
  Table : TruthTable;
```

● Typ *TruthTable* jest związany ze zbiorem tablic o indeksach i elementach typu *boolean*.

● Zmienna *Square* jest tablicą o indeksach typu *Color* i elementach typu *Height*.

● Zmienna *Matrix* jest tablicą o indeksach typu 2..8 i elementach, które są tablicami o indeksach typu 2..8 i elementach typu *integer*.

● Zmienna *Matrix* jest tablicą typu *TruthTable*, tj. tablicą o indeksach typu *boolean*. □

Elementy tablic są reprezentowane w programach za pomocą tzw. *nazw elementów tablic*. Nazwa elementu tablicy składa się z nazwy tablicy, bezpośrednio po której następuje wyrażenie indeksowe ujęte w nawiasy kwadratowe. Wyrażenie indeksowe może być dowolnym wyrażeniem porządkowym takiego samego typu, jaki pod nazwą *typ-indeksów* występuje w opisie typu tablicowego.

Użycie dyrektywy {$*R*+} powoduje generowanie kodu dokonującego kontroli poprawności wyrażeń indeksowych. Kontrola ta polega na upewnieniu się, że wartość wyrażenia indeksowego mieści się w zakresie wyznaczonym przez typ indeksów w opisie typu tablicowego.

Przykład

```
type
  Color = (Red,Green,Blue);
var
  Matrix : array[Color] of boolean;
```

● Zmienna *Matrix* jest tablicą o indeksach typu *Color* i elementach typu *boolean*.

● Poprawnymi nazwami elementów tablicy *Matrix* są m.in.

```
Matrix[Green]
Matrix[pred(Blue)]
Matrix[Color(ord(true))]
```

● W zasięgu dyrektywy kompilatora {$*R*+} taka nazwa elementu tablicy jak np. *Matrix*[*succ*(*Blue*)] zostałaby rozpoznana jako niepoprawna. ■

9.1. TABLICE WIELOWYMIAROWE

Ponieważ elementami tablic mogą być tablice, łatwo jest konstruować tablice dwu- i więcej wymiarowe. Zgodnie ze składnią, definicja typu opisującego tablicę dwuwymiarową przybiera np. postać

```
type
   TwoDim = array[2..5] of array[boolean] of real;
```

Definicja taka może być zapisana w sposób uproszczony

```
type
   TwoDim = array[2..5,boolean] of real;
```

tzn. opisy typu indeksów mogą być ujęte w listę zawartą w nawiasach kwadratowych. Podobne uproszczenie może być zastosowane również w odniesieniu do nazw tablic. W zasięgu deklaracji takiej jak np.

```
var
  Matrix : TwoDim;
```

za poprawne i równoważne uznaje się bowiem takie m.in. nazwy elementów tablicy *Matrix* jak *Matrix*[3][*true*] i *Matrix*[3,*true*].

Przykład

```
type
  Pupils = string[20];
  Class = array[1..30] of Pupils;
var
  School : array[1..20] of Class;
```

- Typ *Class* opisuje zbiór, którego elementami są tablice zawierające wykazy nazwisk uczniów pewnej klasy.
- Zmienna *School* jest tablicą dwuwymiarową, typu

```
array[1..20] of array[1..30] of string[20]
```

w której są przechowywane dane określające nazwiska uczniów każdej z 20 klas danej szkoły.

- Nazwa *School*[3][25] dotyczy ucznia klasy nr 3, występującego na liście uczniów tej klasy na pozycji 25. Nazwa ta jest równoważna nazwie uproszczonej *School*[3,25]. ■

9.2. TABLICE ZNAKOWE

Tablicami znakowymi są tablice jednowymiarowe, o typie indeksów *integer*, których elementy są typu standardowego *char*. Tablice takie mogą być uważane za zmienne łańcuchowe reprezentujące ciągi znaków o stałej długości. Dzięki takiej interpretacji nazwy tablic znakowych i nazwy elementów tych tablic mogą występować w wyrażeniach łańcuchowych wszędzie tam, gdzie mogą występować nazwy zmiennych łańcuchowych. Dotyczy to w szczególności relacji.

Przykład

```
type
   Number = 2..5;
   StrTyp = string[5];
   ArrTyp = array[Number] of char;
```

```
var
   StrVar : StrTyp;
   ArrVar : ArrTyp;
   i : byte;
begin
 for i := 2 to 5 do
   ArrVar[i] := chr(63 + i);
 StrVar := '98765';
 StrVar := Copy(StrVar,2,2) + ArrVar + '/' + ArrVar[4];
```

● Po wykonaniu drugiej instrukcji przypisania zmienna *StrVar* ma wartość '87ABC'.

● W tym samym momencie relacja

```
ArrVar > StrVar
```

ma wartość *true*. ■

9.3. PRZYPISANIA

Zgodnie z zasadą, że zmiennej dowolnego typu może być przypisana dana takiego samego typu, jest możliwe przypisanie, za pomocą jednej instrukcji, całej tablicy zmiennej tablicowej. Dozwolone jest również przypisanie literału znakowego tablicy znakowej, ale tylko wtedy, gdy liczba znaków literału jest równa liczbie znaków tablicy. Zabrania się natomiast przypisywania innych wyrażeń znakowych tablicom. Nic jednak nie stoi na przeszkodzie przypisywaniu wyrażeń łańcuchowych i tablic znakowych zmiennym łańcuchowym.

Przykład

```
type
  Line : string[60];
  Page : array[1..30] of Line;
  Chapter : array[1..50] of Page;
  Word : array[1..5] of char;
var
  Book1,Book2 : array[1..6] of Chapter;
  Arr : Word;
  Brr : array[1..5] of char;
  Str : string[5];
```

● W zasięgu przytoczonych definicji i deklaracji poprawne są m.in. następujące przypisania:

```
Book1 := Book2
Arr := 'Janek'
Str := Arr
```

● Niepoprawne są natomiast przypisania

Arr := *'Jan'* - ponieważ literał liczy za mało znaków
Arr := *Str* - ponieważ *Str* jest wyrażeniem łańcuchowym
Arr := *Brr* - ponieważ *Arr* jest typu *Word*, a *Brr* nie jest typu *Word*. ■

9.4. TABLICE PREDEFINIOWANE

W języku Turbo Pascal dla Elwro 800 Jr predefiniowano dwie tablice o elementach typu *integer*, umożliwiające dostęp do pamięci operacyjnej oraz do portów wejścia/wyjścia.

Tablica Mem

Tablica *Mem* zapewnia dostęp do bajtów pamięci operacyjnej. Indeksem w każdym odwołaniu do tej tablicy jest wyrażenie typu *integer* określające adres bajtu.

Przykład

```
ByteVar := Mem[200]
Mem[300] := $FF
```

● Wykonanie pierwszej instrukcji powoduje przypisanie zmiennej *ByteVar* danej reprezentowanej w bajcie pamięci operacyjnej o adresie 200.

● Wykonanie drugiej instrukcji powoduje umieszczenie danej bajtowej o wartości $FF w bajcie pamięci o adresie 300. ■

Tablica Port

Tablica Port zapewnia dostęp do portów wejścia/wyjścia. Indeksem w każdym odwołaniu do tej tablicy jest wyrażenie typu *integer* określające adres portu. Elementy tablicy *Port* nie mogą występować jako argumenty odwołań do podprogramów, ale mogą występować w wyrażeniach oraz z lewej strony symbolu przypisania. Jeśli występują w wyrażeniu, to reprezentują dane pochodzące z podanego portu. Jeśli występują z lewej strony symbolu przypisania, to następuje przesłanie danej do podanego portu.

Przykład

```
Port[34] := $5F
PortStatus := Port[35]
```

● Wykonanie pierwszej instrukcji powoduje umieszcze-

nie w porcie o adresie 34 danej reprezentowanej przez literał $5F.

● Wykonanie drugiej instrukcji powoduje sięgnięcie do portu o adresie 35 i przypisanie znajdującej się tam danej zmiennej *PortStatus*. ■

W języku Turbo Pascal dla IBM PC predefiniowano analogiczne tablice jak dla Elwro 800 Jr, z tą różnicą, że indeksem tablicy *Mem* jest napis postaci

seg : *ofs*

W napisie tym *seg* reprezentuje numer segmentu, a *ofs* reprezentuje przemieszczenie w segmencie. Ponadto predefiniowano tablice *MemW* i *PortW* umożliwiające dostęp do słów i portów słowowych.

Przykład

Mem[$0020 : $0030] := *'E'*

● Wykonanie przytoczonej instrukcji powoduje zlokalizowanie bajtu o adresie względnym $30, znajdującego się w segmencie o numerze $0020 i umieszczenie w nim danej reprezentowanej przez *'E'*. ■

Każdy z typów rekordowych jest typem złożonym. *Rekord* składa się z ustalonej liczby komponentów, które mogą być różnych typów. Komponenty mogą być zarówno typu prostego, jak i typu złożonego. Dostęp do komponentów, nazywanych również *polami*, odbywa się poprzez *selekcję*. Selekcja polega na określeniu nazwy rekordu, po której następuje znak . (kropka) i identyfikator pola rekordu.

Opis typu rekordowego składa się ze słowa kluczowego **record**, bezpośrednio po którym następuje wykaz pól rekordu i słowo kluczowe **end**. Każdy element wykazu ma postać deklaracji pola. Deklaracje pól są oddzielone średnikami, a ostatnim elementem wykazu może być część wariantowa rekordu. Część wariantowa składa się ze słowa kluczowego **case**, bezpośrednio po którym następuje deklaracja pola wyróżnikowego albo opis typu wyróżnikowego, słowo kluczowe **of**, a bezpośrednio po nim wykaz wariantów. Każdy element wykazu wariantów składa się z listy etykiet wyboru zakończonej dwukropkiem i ujętego w nawiasy okrągłe wykazu pól wariantu. Poszczególne elementy wykazu wariantów są oddzielone średnikami.

Składnia

opis-typu-rekordowego:
  record *wykaz-pól-rekordu* **end**
wykaz-pól-rekordu:
  deklaracja-pola
  część-wariantowa-rekordu
deklaracja-pola:
  lista-nazw-pól : *opis-typu*
nazwa-pola:
  identyfikator
część-wariantowa-rekordu:

```
    case deklaracja-pola-wyróżnikowego of wykaz-wariantów
    case opis-typu-wyróżnikowego of wykaz-wariantów
deklaracja-pola-wyróżnikowego:
    nazwa-pola-wyróżnikowego : opis-typu-wyróżnikowego
nazwa-pola-wyróżnikowego:
    identyfikator
opis-typu-wyróżnikowego:
    identyfikator-typu-porządkowego
wykaz-wariantów:
    lista-etykiet-wyboru : ( wykaz-pól-wariantu )
etykieta-wyboru:
    literał
    nazwa-literału
```

Przykłady

```
type
  Days = 1..31;
  Date = record
           Day : Days;
           Month : 1..12;
           Year : 1900..1999
         end;
  Person = record
             Name : string[20];
             BirthPlace : string[30];
             case Sex : (male,female,alien) of
               male,female : (BirthDate : Date);
               alien : (BirthDate : real;
                        BodyData : record
                                     H : integer;
                                     W : real
                                   end)
           end;
var
  MyBirthDay : Date;
  EarthMan,Galaxian : Person;
```

● Typ *Date* jest związany ze zbiorem rekordów, których pola są typu *Days*, 1..12 i 1900..1999.

● Typ *Person* jest związany ze zbiorem rekordów z wariantami. Wyróżnikiem wariantów jest pole *Sex* typu (*male,female, alien*). Jeśli wyróżnik *Sex* ma wartość *male* albo *female*, to rekord typu *Person* składa się z pól typu **string**[20], **string** [30], (*male,female,alien*), 1..31, 1..12 i 1900..1999. Jeśli wyróżnik *Sex* ma wartość *alien*, to rekord typu *Person* składa się z pól typu **string**[20], **string**[30], (*male,female,alien*), *real*, *integer* i *real*.

• Zmienna *MyBirthDay* jest rekordem składającym się z pól *Day*, *Month* i *Year*.

• Zmienne *EarthMan* i *Galaxian* są rekordami typu *Person*. Wybór wariantu rekordów odbywa się przez przypisanie polu wyróżnikowemu danej typu (*male,female,alien*).

• Pole wyróżnikowe zmiennej *EarthMan* jest identyfikowane przez selekcję *EarthMan.Sex*, a pole wyróżnikowe zmiennej *Galaxian* przez selekcję *Galaxian.Sex*.

• Po wykonaniu przypisania

Galaxian.Sex := alien

jest dozwolone odwoływanie się do komponentów wariantu *alien*. W szczególności poprawne jest wówczas przypisanie

Galaxian.BodyData.H := 20

w którym operator selekcji . (kropka) zastosowano najpierw do komponentów rekordu *Galaxian*, a następnie do komponentów rekordu *Galaxian.BodyData*. ■

10.1. INSTRUKCJA WITH

Odwoływanie się do komponentów rekordu wymaga podania nie tylko nazwy pola, ale również nazwy samego rekordu. Ponieważ w pewnych przypadkach znacznie wydłuża to tekst programu, wygodnie jest posłużyć się *instrukcją wiążącą with* "odsłaniającą" wnętrze definicji typu rekordowego.

W ogólnym przypadku instrukcja with składa się ze słowa kluczowego **with**, bezpośrednio po którym następuje lista nazw rekordów, słowo kluczowe **do** i dowolna instrukcja. W jej obrębie selekcja pola rekordu wymienionego w liście może być zastąpiona samą nazwą pola. Przyjmuje się z definicji, że instrukcja o postaci

with *a,b, ...,z* **do** *i*

jest równoważna instrukcji

```
with a do
  with b do
    ...
    with z do i
```

Składnia

instrukcja-with:
  with *lista-nazw-rekordów* **do** *instrukcja*
nazwa-rekordu:
  identyfikator
  nazwa-elementu-tablicy
  nazwa-pola

Przykład

```
var
 Alfa : record
          Beta : record
                   Gamma : char;
                   Delta : real
                 end;
          Gamma : integer
        end;
 Beta : array[boolean] of record
                            Chi,Tau : real
                          end;
```

- W zasiegu przytoczonej deklaracji poprawna jest instrukcja

```
with Alfa.Beta,Beta[true] do
  Delta := Tau
```

wykonywana tak jak instrukcja

```
Alfa.Beta.Delta := Beta[true].Tau
```

- Poprawna jest także instrukcja

```
with Alfa.Beta,Alfa do begin
   Delta := 23.4;
   Writeln(Gamma)
 end
```

wykonywana jak instrukcja

```
begin
   Alfa.Beta.Delta := 23.4;
   Writeln(Alfa.Gamma)
 end
```

- Należy zwrócić uwagę, że kolejność nazw rekordów na liście instrukcji with jest istotna, gdyż instrukcja taka jak

```
with Alfa,Alfa.Beta do begin
   Delta := 23.4;
   Writeln(Gamma)
 end
```

jest wykonywana tak jak instrukcja

```
begin
   Alfa.Beta.Delta := 23.4;
   Writeln(Alfa.Beta.Gamma)
 end
```

a nie jak instrukcja

```
begin
   Alfa.Beta.Delta := 23.4;
   Writeln(Alfa.Gamma)
 end
```

■

10.2. TRAKTOWANIE WARIANTOW JAKO UNII

Jeśli rekord zawiera część-wariantową, to w reprezentacji wewnętrznej przyjętej w Turbo Pascalu, pierwsze pole każdego wariantu zaczyna się od tego samego bajtu pamięci. Biorąc ponadto pod uwagę, że deklaracja pola wyróżnikowego części wariantowej może zostać uproszczona do opisu typu, łatwo jest przekształcić opis rekordu w opis unii. Należy jedynie uwzględnić, iż w odróżnieniu od operacji na rekordach operacje na uniach wymagają znajomości sposobu reprezentowania komponentów, a przynajmniej sposobu rozmieszczenia i rozmiaru każdego z nich.

Przykład

```
var
  Word : record
           case boolean of
             false : (LowByte,HighByte : byte);
             true : (FullWord : integer)
           end
```

- W zasięgu przytoczonej deklaracji dopuszczalne jest zastąpienie instrukcji

```
with Word do
  FullWord := FullWord and $FF00
```

instrukcją

```
with Word do
  LowByte := 0
```

- Poprawność przytoczonego postępowania wynika z faktu, że mniej znaczący bajt danej typu *integer* zajmuje w pamięci operacyjnej bajt o niższym adresie, pola struktury zaś występują w pamięci operacyjnej w takiej samej kolejności w jakiej zostały zadeklarowane (tu *LowByte* przed *HighByte*). ■

10.3. PRZYPISANIA REKORDOW

Analogicznie do przypisywania zmiennym tablicowym całych tablic jest dozwolone przypisywanie zmiennym rekordowym całych rekordów. Wymaga się jedynie, aby typ zmiennej rekordowej był identyczny z typem przypisywanego jej rekordu.

Przykład

```
type
  Name = record
           LastName : string[32];
```

```
            Names : array[(fst,snd,trd)] of string[8]
          end;
    var
      JohnName,EwaName : Name;
      ...
    EwaName := JohnName;                                    ■
```

Każdy z typów mnogościowych jest typem złożonym. *Mnogość* składa się z wybranego podzbioru pewnego zbioru potęgowego. Zbiorem podstawowym zbioru potęgowego może być dowolny zbiór elementów typu porządkowego. Każdy element mnogości jest jednym z takich właśnie elementów.

Dwie dane mnogościowe są uznawane za równe wtedy i tylko wtedy, gdy składają się z takich samych elementów. Jeśli wszystkie elementy jednej mnogości są elementami drugiej, to mówi się, że pierwsza mnogość zawiera się w drugiej.

Opis typu mnogościowego składa się ze słowa kluczowego **set**, bezpośrednio po którym następuje słowo kluczowe **of** i opis typu podstawowego. Wymaga się, aby typ porządkowy pełniący rolę typu podstawowego nie zawierał więcej niż 256 elementów oraz aby rezultat funkcji *ord* dla każdego z tych elementów mieścił się w przedziale domkniętym [0,255].

Składnia

opis-typu-mnogościowego:
set of *typ-podstawowy*
typ-podstawowy:
typ-porządkowy

Przykłady

```
type
  DaysOfWeek : set of (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
  Chars : set of char;
  SmallLetters : set of 'a'..'z';
  DayNumbers : set of 1..31;
```

- Wartością danej typu *DaysOfWeek* może być dowolny, w tym pusty, podzbiór zbioru nazw dni tygodnia.
- Wartością danej typu *Chars* może być dowolny podzbiór

zbioru znaków ASCII.

● Wartością danej typu *SmallLetters* może być dowolny podzbiór zbioru małych liter alfabetu ASCII.

● Wartością danej typu *DayNumbers* może być dowolny podzbiór zbioru liczb całkowitych z przedziału domkniętego [1,31]. ■

11.1. LITERAŁY MNOGOSCIOWE

Literał mnogościowy składa się z zawartej w nawiasach kwadratowych listy literałów, z których każdy jest tego samego typu porządkowego. Lista ta może być pusta, może zawierać powtórzenia elementów oraz może zawierać konstrukcje o postaci *min..max*, w których *min* i *max* są literałami tego samego typu. W tym ostatnim przypadku napis *min..max* jest równoważny liście wszystkich literałów od *min* do *max* włącznie. Jeśli *min>max*, to napis ten opisuje podzbiór pusty.

Należy nadmienić, że wartość literału mnogościowego nie zależy od kolejności tworzących go literałów typu podstawowego, a wielokrotne ich wystąpienia nie mają wpływu na wartość literału mnogościowego.

Przykład. Kilka poprawnych i niepoprawnych literałów mnogościowych

Literały poprawne: ['P','a','s','c','a','l']
[4,9,5,0,8,6]
[*true*,*false*]
[]
[20..30,50,70..80]

Napisy, które nie są literałami mnogościowymi:

[*true*,2] – różne typy podstawowe
[200..300] – *ord*(300)>255
[4.0,5.0] – typ podstawowy nie jest typem porządkowym ■

11.2. WYRAZENIA

Wyrażenia mnogościowe składają się z odwołań do zmiennych mnogościowych, odwołań do literałów i konstruktorów mnogościowych oraz operatorów sumy, różnicy i przecięcia.

Konstruktory mnogościowe mają postać zbliżoną do literałów mnogościowych, tyle że elementami ich list mogą być nie tylko literały typu podstawowego, ale również dowolne wyrażenia tego typu.

Operator sumy mnogości jest zapisywany za pomocą znaku + (plus), *operator różnicy* za pomocą znaku - (minus), a *operator przecięcia* za pomocą znaku * (gwiazdka).

Sumą mnogości jest mnogość składająca się z wszystkich elementów obu składników. *Różnicą mnogości* jest mnogość składająca się z tych wszystkich elementów odjemnej, które nie są elementami odjemnika. *Przecięciem mnogości* jest mnogość składająca się z wszystkich elementów wspólnych czynników przecięcia.

Poza operacjami mnogościowymi, dane typu mnogościowego mogą występować w relacjach:

a = *b*	prawdziwej, jeśli *a* i *b* są zbiorami tych samych elementów;
a <> *b*	prawdziwej, jeśli *a* i *b* są różnymi zbiorami elementów;
a <= *b*	prawdziwej, jeśli *a* jest pustym zbiorem elementów albo gdy każdy element należący do *a* należy także do *b*;
a >= *b*	prawdziwej, gdy *b* jest pustym zbiorem elementów albo gdy każdy element należący do *b* należy także do *a*;
e **in** *a*	prawdziwej, gdy element typu podstawowego *e* jest elementem zbioru elementów *a*.

Przykłady. W zasięgu deklaracji zmiennej *SetVar*, której przypisano daną mnogościową reprezentowaną przez literał [*Red*,*Green*]

```
type
   Colors = (Red,Green,Blue);
   Paint = set of Colors;
var
   SetVar : Paint;
...
SetVar := [Red,Green];
```

prawdziwe jest zestawienie:

Wyrażenie	Rezultat
SetVar = [*Red*..*Blue*] - [*Blue*]	*true*
SetVar <> []	*true*
SetVar <= [*Red*..*succ*(*Green*)]	*true*
SetVar >= [*Blue*..*Green*]	*true*
SetVar * [*Red*] = [*Red*]	*true*
[*Green*] **in** *SetVar*	*true*
[] **in** *SetVar*	*true* ■

11.3. PRZYPISANIA MNOGOSCI

Analogicznie do przypisania zmiennym tablicowym całych tablic oraz zmiennym rekordowym całych rekordów, jest dozwolone przypisywanie zmiennym mnogościowym mnogości. Wymaga się jedynie, aby typ zmiennej mnogościowej był identyczny z typem przypisywanej jej danej mnogościowej.

Przykład

```
var
  SetVar : array[2..5,boolean] of set of 20..30;
...
SetVar[3,false] := [5*5 .. pred(29)];
```

● Prawą stronę instrukcji przypisania stanowi konstruktor mnogościowy. Konstruktor ten jest równoważny literałowi mnogościowemu [25,26,27,28]. ■

12. TYPY PLIKOWE

Każdy z typów plikowych jest typem złożonym. *Plik* składa się z komponentów, z których każdy jest takiego samego typu. Komponenty pliku są nazywane jego elementami. W odróżnieniu od typu tablicowego liczba elementów pliku nie jest ustalona w deklaracji pliku, lecz jest uzależniona od przebiegu wykonywania programu, a w szczególności od tego, z jakim zbiorem danych rozpatrywany plik zostanie skojarzony.

Należy podkreślić, że obiekt nazywany plikiem jest jedynie abstrakcyjnym modelem fizycznego zbioru danych występującego najczęściej poza programem. Fizyczne zbiory danych - nazywane tu krótko *zbiorami* - mogą rezydować w pamięci zewnętrznej komputera, mogą zajmować część jego pamięci operacyjnej, a także mogą być utożsamiane ze strumieniami danych wprowadzanych i wyprowadzanych za pomocą urządzeń zewnętrznych.

Możliwość przetwarzania różnych zbiorów danych za pomocą tego samego pliku, po uprzednim skojarzeniu pliku ze zbiorem, stanowi znaczne ułatwienie programowania, ponieważ ogranicza zadanie programującego do koncentrowania się na istotnych elementach algorytmu przetwarzania bez wnikania w działania specyficzne dla danego systemu operacyjnego oraz w sposób reprezentowania danych w pamięci zewnętrznej. Z tego względu programowanie na poziomie abstrakcji nazywanej plikiem sprowadza się głównie do otwarcia pliku, wykonania operacji na jego elementach oraz do zamknięcia pliku.

Opis typu plikowego składa się ze słowa kluczowego file bezpośrednio po którym następuje słowo kluczowe of, a za nim opis typu elementów pliku. Elementami pliku mogą być obiekty dowolnych typów prostych, a także dowolne agregaty z wyjątkiem plików.

Składnia

opis-typu-plikowego:
file of *typ-elementów-pliku*
typ-elementów-pliku:
opis-typu

Przykłady

```
type
  Measurements : file of real;
  Persons : file of
              record
                Name : string[20];
                Sex : (male,female);
                Age : 18..65
              end;
var
  Experiment : Measurements;
  Population : Persons;
  Complexes : file of record
                        Re,Im : real
                      end;
  Arrays : file of array[2..6,4..8,boolean] of byte;
```

● Typ *Measurements* jest związany ze zbiorem sekwencji danych typu *real*.

● Typ *Persons* jest związany ze zbiorem sekwencji składających się z rekordów o komponentach typu **string**[20], (*male*,*female*) i 18..65.

● Zmienna *Experiment* jest zmienną plikową typu *Measurement*.

● *Complexes* jest zmienną plikową reprezentującą plik związany ze zbiorem sekwencji par danych typu *real*.

● Zmienna *Arrays* jest zmienną plikową reprezentującą plik związany ze zbiorem sekwencji tablic typu **array**[2..6,4..8,*boolean*] **of** *byte*. ■

12.1. PODPROGRAMY

Operacje na plikach mogą być wykonywane jedynie za pomocą wywołań funkcji i procedur. Każda z takich operacji musi być poprzedzona skojarzeniem pliku ze zbiorem danych, a jeśli dotyczy elementów pliku, to musi być także poprzedzona otwarciem pliku. Skojarzenie pliku ze zbiorem odbywa się za pomocą procedury *Assign*, a otwarcie pliku za pomocą procedur *Reset* i *Rewrite*. Użycie procedury *Reset* nie implikuje, że plik jest

otwierany wyłącznie w celu wprowadzania elementów zbioru, a użycie procedury *Rewrite* nie implikuje, że plik jest otwierany wyłącznie w celu ich wyprowadzania. Przed zakończeniem wykonywania programu każdy plik powinien zostać zamknięty. Służy do tego procedura *Close*. Samo zakończenie wykonywania programu nie pociąga za sobą zamknięcia nie zamkniętych jeszcze plików.

Bezpośrednio po otwarciu plik znajduje się w pozycji początkowej. Zmiana pozycji pliku może być uzyskana za pomocą procedury *Seek*. Aktualny rozmiar pliku może być określony za pomocą funkcji *FileSize*, a jego aktualna pozycja za pomocą funkcji *FilePos*. Jeśli plik zostanie ustawiony w pozycji pośredniej, tj. między pozycją początkową a końcową, to każde wykonanie procedury *Write* spowoduje zmianę najbliższego elementu pliku. Zmiana ta nie będzie miała żadnego wpływu na stan pozostałych elementów pliku. Z tego względu pliki o elementach ustalonego typu mogą reprezentować zbiory o organizacji sekwencyjno-wyrywkowej.

Procedura Assign

Wywołanie: *Assign(FileVar,StrExp)*

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej, a *StrExp* jest wyrażeniem łańcuchowym, Wymaga się, aby plik reprezentowany przez *FileVar* nie był otwarty.

Wykonanie procedury *Assign* powoduje skojarzenie pliku reprezentowanego przez zmienną *FileVar* ze zbiorem danych o nazwie określonej przez wartość danej reprezentowanej przez wyrażenie *StrExp*.

Przykład

```
var
  Results : file of real;
...
Assign(Results,'A:TESTS.DOC');
```

- Zmienna plikowa *Results* reprezentuje plik o elementach typu *real*.
- Wykonanie procedury *Assign* powoduje skojarzenie pliku reprezentowanego przez zmienną *Results* ze zbiorem danych *TESTS.DOC* znajdującym się na dyskietce umieszczonej w stacji dyskowej *A*.
- Wykonanie procedury *Assign* nie powoduje otwarcia pliku reprezentowanego przez zmienną *Results*. ■

Procedura Reset

Wywołanie: *Reset(FileVar)*

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej. Wymaga się, aby przed wywołaniem procedury *Reset* plik reprezentowany przez zmienną *FileVar* był skojarzony z istniejącym zbiorem danych.

Wykonanie procedury *Reset* powoduje otwarcie pliku reprezentowanego przez zmienną *FileVar*. Bezpośrednio po otwarciu plik zostaje ustawiony w pozycji początkowej, tj. tuż przed jego pierwszym elementem.

Przykład

```
var
  InpFile : file of record
                      Name: string[30];
                      Income : real
                    end;
Assign(InpFile,'INCOME.DOC');
Reset(InpFile);
```

- Wykonanie procedury *Assign* powoduje skojarzenie pliku reprezentowanego przez zmienną plikową *InpFile* ze zbiorem *INCOME.DOC* znajdujacym się na dyskietce umieszczonej w domniemanej stacji dyskowej.
- Wykonanie procedury *Reset* powoduje otwarcie pliku reprezentowanego przez zmienną *InpFile*.
- Otwarcie pliku za pomocą procedury *Reset* nie wyklucza możliwości przyszłego odwoływania się do niego za pomocą procedur *Seek* i *Write*. ■

Procedura Rewrite

Wywołanie: *Rewrite(FileVar)*

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej. Wymaga się, aby przed wywołaniem procedury *Rewrite* plik reprezentowany przez zmienną *FileVar* był skojarzony ze zbiorem danych.

Wykonanie procedury *Rewrite* powoduje otwarcie pliku reprezentowanego przez zmienną *FileVar*. Bezpośrednio po otwarciu plik zostaje ustawiony w pozycji początkowej, tj. tuż przed jego pierwszym elementem.

Jeśli przed wykonaniem procedury *Rewrite* zbiór skojarzony z plikiem nie istniał, to zostanie utworzony. Jeśli istniał, to zostanie usunięty i utworzony ponownie. W obu przypadkach zostanie utworzony zbiór pusty.

Przykład

```
var
  OutFile : file of array[1..20] of byte;
...
Assign(OutFile,'B:TESTS.OUT');
Rewrite(OutFile);
```

• Wykonanie procedury *Assign* powoduje skojarzenie pliku reprezentowanego przez zmienną plikową *OutFile* ze zbiorem *TESTS.OUT* znajdującym się na dyskietce umieszczonej w stacji dyskowej *B*.

• Wykonanie procedury *Rewrite* powoduje otwarcie pliku reprezentowanego przez zmienną *OutFile*. Po dokonaniu otwarcia plik reprezentowany przez tę zmienną jest pusty.

• Otwarcie pliku za pomocą procedury *Rewrite* nie wyklucza możliwości odwoływania się do niego za pomocą wywołań procedur *Seek* i *Read*. ■

Procedura Read

Wywołanie: *Read(FileVar,VarList)*

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej, a *VarList* jest nazwą zmiennej albo listą nazw zmiennych. Wymaga się, aby plik reprezentowany przez zmienną *FileVar* był otwarty. Zabrania się, aby jakakolwiek zmienna *VarList* była innego typu niż typ elementów pliku.

Wykonanie procedury *Read* z listą nazw zmiennych jest równoważne wykonaniu ciągu procedur *Read* z nazwami tych zmiennych. Wykonanie procedury *Read* z nazwą zmiennej powoduje wprowadzenie z pliku jednego elementu i przypisanie go zmiennej o podanej nazwie.

Przykład

```
type
  ElmType = record
              Re,Im : real
            end;
var
  InpFile : file of ElmType;
  ArrVar : array[boolean,2..4] of ElmType;
...
Assign(InpFile,'COMPLEX.DOC');
Reset(InpFile);
Read(InpFile,ArrVar[true,3]);
```

• Wykonanie procedury *Assign* powoduje skojarzenie pli-

ku reprezentowanego przez zmienną plikową *InpFile* ze zbiorem *COMPLEX.DOC*.

● Wykonanie procedury *Reset* powoduje otwarcie pliku reprezentowanego przez zmienną *InpFile*.

● Wykonanie procedury *Read* powoduje wprowadzenie z pliku reprezentowanego przez *InpFile* jednego elementu i przypisanie go elementowi *ArrVar[true,3]* tablicy *ArrVar*. ■

Procedura Write

Wywołanie: *Write(FileVar,VarList)*

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej, a *VarList* jest nazwą zmiennej albo listą nazw zmiennych. Wymaga się, aby plik reprezentowany przez zmienną *FileVar* był otwarty. Zabrania się, aby jakakolwiek zmienna *VarList* była innego typu niż typ elementów pliku.

Wykonanie procedury *Write* z listą nazw zmiennych jest równoważne wykonaniu ciągu procedur *Write* z nazwami tych zmiennych. Wykonanie procedury *Write* z nazwą zmiennej powoduje wyprowadzenie do pliku reprezentowanego przez *FileVar* danej przypisanej tej zmiennej.

Przykład

```
type
  ElmType = record
              Re,Im : real
            end;
var
  OutFile : file of ElmType;
  ArrVar : array[boolean] of ElmType;
...
Assign(OutFile,'COMPLEX.RES');
Rewrite(OutFile);
Write(OutFile,ArrVar[false],ArrVar[true]);
```

● Wykonanie procedury *Assign* powoduje skojarzenie pliku reprezentowanego przez zmienną plikową *OutFile* ze zbiorem *COMPLEX.RES*.

● Wykonanie procedury *Rewrite* powoduje otwarcie pliku reprezentowanego przez zmienną *OutFile*.

● Wykonanie procedury *Write* powoduje wyprowadzenie do pliku reprezentowanego przez *OutFile* danych przypisanych elementom *ArrVar[false]* i *ArrVar[true]* tablicy *ArrVar*.

● Zastąpienie przytoczonej instrukcji *Write* instrukcją

```
Write(OutFile,ArrVar);
```

byłoby niepoprawne, ponieważ sama nazwa tablicy nie stanowi wyszczególnienia jej elementów. ■

Procedura Seek

Wywołanie: *Seek(FileVar,PosExp)*

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej, a *PosExp* jest wyrażeniem typu *integer*. Wymaga się, aby plik reprezentowany przez zmienną *FileVar* był otwarty. Zabrania się, aby wartość wyrażenia *PosExp* wykraczała poza przedział domknięty [0,*FileSize(FileVar)*].

Wykonanie procedury *Seek* powoduje ustawienie pliku w takiej pozycji, aby najbliższym elementem pliku był element o numerze porządkowym określonym przez wartość wyrażenia *PosExp*. Jeśli wyrażenie to ma wartość 0, to plik zostanie ustawiony w pozycji początkowej, a jeśli ma wartość *FileSize(FileVar)*, to zostanie ustawiony w pozycji końcowej.

Przykład

```
type
  Name = string[5];
var
  FileVar : file of Name;
  ElmVar1,ElmVar2 : Name;
  Len,Pos : integer;
begin
  Assign(FileVar,'FAMILY');
  Reset(FileVar);
  Len := FileSize(FileVar) - 1;
  for Pos := 0 to Len shr 1 do begin
    Seek(FileVar,Pos);
    Read(FileVar,ElmVar1);
    Seek(FileVar,Len - Pos);
    Read(FileVar,ElmVar2);
    Seek(FileVar,Pos);
    Write(FileVar,ElmVar2);
    Seek(FileVar,Len - Pos);
    Write(FileVar,ElmVar1)
  end;
 Close(FileVar)
end.
```

● Wykonanie przytoczonego programu powoduje odwrócenie porządku elementów zbioru *FAMILY*.

● Otwarcie pliku reprezentowanego przez zmienną *FileVar* zrealizowano za pomocą procedury *Reset*, ponieważ użycie procedury *Rewrite* spowodowałoby utratę zawartości zbioru *FAMILY*.

■

Procedura Close

Wywołanie: *Close(FileVar)*

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej. Nie wymaga się, aby plik reprezentowany przez zmienną *FileVar* był otwarty.

Wykonanie procedury *Close* powoduje zamknięcie pliku reprezentowanego przez zmienną *FileVar*. Jeśli przed wykonaniem tej procedury plik nie był otwarty, to jego stan nie zmienia się.

Należy nadmienić, że zakończenie wykonywania programu nie implikuje domniemanych wywołań procedury *Close*.

Przykład

```
var
  NewFile : file of boolean;
begin
  Assign(NewFile,'EMPTY');
  Rewrite(NewFile);
  Close(NewFile)
end.
```

● Wykonanie przytoczonego programu pozostawia na dyskietce, umieszczonej w domniemanej stacji dyskowej, pusty zbiór *EMPTY*.

● W przyszłości zbiór ten może być wypełniony elementami typu *boolean*. ■

Procedura Erase

Wywołanie: *Erase(FileVar)*

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej. Zaleca się, aby plik reprezentowany przez zmienną *FileVar* nie był otwarty.

Wykonanie procedury *Erase* powoduje usunięcie zbioru danych skojarzonego z plikiem reprezentowanym przez zmienną *FileVar*.

Przykład

```
var
  KillFile : file of byte;
begin
  Assign(KillFile,'SECRET.DOC');
  Erase(KillFile)
end.
```

● Wykonanie przytoczonego programu powoduje usunięcie z

dyskietki, umieszczonej w domniemanej stacji dyskowej, zbioru *SECRET.DOC*.

● Operacja dotyczy pliku skojarzonego ze zbiorem danych, ale nie otwartego.

● Typ elementów pliku reprezentowanego przez zmienną plikową *KillFile* jest bez znaczenia. ■

Procedura Rename

Wywołanie: *Rename(FileVar,StrExp)*

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej, a *StrExp* jest wyrażeniem łańcuchowym. Zabrania się, aby plik reprezentowany przez zmienną *FileVar* był otwarty.

Wykonanie procedury *Rename* powoduje zmianę nazwy zbioru danych skojarzonego z plikiem reprezentowanym przez *FileVar* na nazwę określoną za pomocą wyrażenia łańcuchowego *StrExp*. Wymaga się, aby ciąg znaków reprezentowany przez *StrExp* był poprawną nazwą zbioru, pozbawioną określenia stacji dyskowej, a ponadto aby nie był nazwą zbioru już istniejącego. Zabrania się, aby plik reprezentowany przez *FileVar* był otwarty.

Przykład

```
var
  RenFile : file of byte;
begin
  Assign(RenFile,'SECRET.DOC');
  Rename(RenFile,'SECRET.BAK')
end.
```

● Wykonanie przytoczonego programu powoduje zmianę nazwy zbioru *SECRET.DOC* na nazwę *SECRET.BAK*.

● Typ elementów pliku reprezentowanego przez zmienną plikową jest bez znaczenia. ■

Funkcja Eof

Wywołanie: *Eof(FileVar)*

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej. Wymaga się, aby plik reprezentowany przez *FileVar* był otwarty.

Rezultatem funkcji *Eof* jest dana o wartości *true* - jeśli plik reprezentowany przez zmienną *FileVar* znajduje się w pozycji końcowej, albo dana o wartości *false* - w przeciwnym razie.

Przykład

```
var
  FileVar : file of byte;
```

```
begin
  Assign(FileVar,'USELESS.DOC');
  Rewrite(FileVar);
  Writeln(Eof(FileVar))
end.
```

● Bezpośrednio po wykonaniu procedury *Rewrite* plik znajduje się zarówno w pozycji początkowej, jak i końcowej.

● Wykonanie instrukcji

```
Writeln(Eof(FileVar))
```

powoduje wyprowadzenie napisu *TRUE*. ■

Funkcja FileSize

Wywołanie: *FileSize(FileVar)*

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej. Wymaga się, aby plik reprezentowany przez *FileVar* był otwarty.

Rezultatem funkcji *FileSize* jest dana typu *integer*. Wartością tej danej jest liczba elementów pliku reprezentowanego przez zmienną *FileVar*.

Przykład

```
var
  FileVar : file of 2..4;
begin
  Assign(FileVar,'USELESS.DOC');
  Rewrite(FileVar);
  Writeln(FileSize(FileVar))
end.
```

● Bezpośrednio po wykonaniu procedury *Rewrite* plik reprezentowany przez zmienną *FileVar* jest pusty.

● Ponieważ plik pusty zawiera 0 elementów, wykonanie przytoczonego programu powoduje wyprowadzenie liczby 0. ■

Funkcja FilePos

Wywołanie: *FilePos(FileVar)*

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej. Wymaga się, aby plik reprezentowany przez *FileVar* był otwarty.

Rezultatem funkcji *FilePos* jest dana typu *integer*. Wartością tej danej jest numer pozycji pliku reprezentowanego przez zmienną *FileVar*. Numerem pozycji początkowej jest 0, a numerem pozycji końcowej jest *FileSize(FileVar)*.

Przykład

```
var
  FileVar : file of (Red,Green,Blue);
begin
  Assign(FileVar,'COLORS');
  Reset(FileVar);
  Seek(FileSize(FileVar));
  Writeln(FilePos = 0)
end.
```

- Jeśli zbiór *COLORS* jest pusty, to wykonanie przytoczonego programu spowoduje wyprowadzenie napisu *TRUE*.
- W przeciwnym razie nastąpi wyprowadzenie napisu *FALSE*

■

12.2. PLIKI TEKSTOWE

W odróżnieniu od pozostałych plików pliki tekstowe nie składają się z sekwencji identycznych elementów tego samego typu, lecz składają się z wierszy podzielonych na znaki. Każdy wiersz pliku tekstowego jest zakończony parą znaków CR/LF (carriage return/line feed). Bezpośrednio po ostatnim wierszu pliku występuje znak Ctrl-Z.

Ponieważ wiersze pliku mogą być różnej długości, przetwarzanie pliku może być jedynie sekwencyjne, plik zaś może zostać otwarty tylko do wyprowadzania (*Rewrite*) albo tylko do wprowadzania (*Reset*).

Opis typu plikowego tekstowego składa się z predefiniowanego identyfikatora *text*.

Składnia

opis-typu-plikowego-tekstowego:
  text

Przykład

```
type
  TextType = text;
var
  OutFile : TextType;
  InpFile : text;
```

- Typ *TextType* jest związany ze zbiorem sekwencji wierszy podzielonych na znaki i zakończonych znakami CR/LF.
- Zmienne plikowe *OutFile* i *InpFile* reprezentują pliki tekstowe. ■

W języku Turbo Pascal komunikowanie się z takimi urządzeniami zewnętrznymi jak konsole, terminale, drukarki i modemy odbywa się za pomocą plików tekstowych. Pliki te stanowią zatem modele fizycznych zbiorów danych dostępnych za pomocą tych urządzeń.

Urządzenia zewnętrzne wymienionych tu typów mają swoje symboliczne oznaczenia, składające się z trzyliterowej mnemoniki i znaku : (dwukropek).

CON: - *konsola*

Konsola jest urządzeniem wejściowo-wyjściowym. Elementem wyjściowym konsoli jest ekran monitora, a elementem wejściowym - klawiatura. Wprowadzenie danych z konsoli jest *buforowane*. Oznacza to, że dane są wprowadzane z konsoli pełnymi wierszami i dopiero po wprowadzeniu całego wiersza, stanowiące go znaki podlegają przetwarzaniu. Ponieważ każdy wiersz jest kończony znakiem CR, możliwe jest wprowadzenie z konsoli ciągu znaków, a następnie - ale przed wprowadzeniem znaku CR - poddanie go edycji. Edycja może być realizowana za pomocą następujących znaków.

Ctrl-H:
Usunięcie znaku znajdującego się z lewej strony kursora i przesunięcie kursora o jedną pozycję w lewo.

Ctrl-X:
Usunięcie wszystkich znaków znajdujących się z lewej strony kursora i przesunięcie kursora do pierwszej kolumny wiersza.

Ctrl-D:
Wstawienie na pozycję kursora jednego znaku z poprzedniego wiersza i przesunięcie kursora o jedną pozycję w prawo.

Ctrl-R:
Wstawienie, począwszy od pozycji kursora, pozostałych znaków poprzedniego wiersza i przesunięcie kursora w prawo za ostatni wstawiony znak.

CR, Ctrl-M:
Zakończenie wprowadzania wiersza i umieszczenie w buforze wejściowym pary znaków CR/LF.

Ctrl-Z:
Zakończenie wprowadzania wiersza i umieszczenie w buforze wejściowym znaku Ctrl-Z.

Należy nadmienić, że rozmiar bufora wejściowego konsoli jest określony przez predefiniowaną zmienną *BufLen*. Zarówno

maksymalną, jak i domniemaną wartością tej zmiennej jest 127. Ustalenie nowej wartości zmiennej *BufLen* jest skuteczne jedynie w odniesieniu do najbliższej instrukcji wprowadzania. Bezpośrednio po wykonaniu każdej takiej instrukcji zmienna *BufLen* otrzymuje ponownie wartość 127.

TRM: - *terminal*

Terminal jest urządzeniem wejściowo-wyjściowym. Elementem wyjściowym terminala jest ekran monitora, a elementem wejściowym - klawiatura. W odróżnieniu od konsoli wprowadzanie danych z terminala nie jest buforowane. Oznacza to, że każdy wprowadzany znak natychmiast podlega przetwarzaniu. Jednocześnie ze skierowaniem do przetwarzania znak jest wyprowadzany na ekran. Spośród znaków sterujących dotyczy to jednak tylko znaku CR, który jest wyprowadzany jako para CR/LF.

KBD: - *klawiatura*

Klawiatura jest urządzeniem wejściowym. Znaki wprowadzane z urządzenia KBD: pochodzą z elementu wejściowego konsoli, nie podlegają buforowaniu i nie są wyprowadzane na ekran.

LST: - *drukarka*

Drukarka jest urządzeniem wyjściowym. Wyprowadzane znaki nie podlegają buforowaniu w systemie, ale mogą być buforowane w drukarce.

Skojarzenie pliku z urządzeniem logicznym może być dokonane za pomocą procedury *Assign*, w której wywołaniu jest podawana nazwa zmiennej plikowej oraz wyrażenie łańcuchowe określające nazwę urządzenia logicznego. W odróżnieniu od skojarzenia pliku ze zbiorem, skojarzenie pliku z urządzeniem logicznym pociąga za sobą niejawne otwarcie pliku. Z tego względu posłużenie się procedurami *Reset* i *Rewrite* jest zbyteczne, a ich wykonanie, podobnie jak wykonanie procedury *Close*, nie ma żadnych skutków. Błędne jest natomiast posłużenie się takimi procedurami jak *Erase* i *Rename*, gdyż można je stosować jedynie w odniesieniu do zbiorów danych w pamięci dyskowej.

Przykład

```
var
  Console : text;
...
Assign(Console,'CON:');
```

● Wykonanie procedury *Assign* spowoduje skojarzenie pliku tekstowego reprezentowanego przez zmienną plikową *Console* z urządzeniem logicznym CON:, tj. konsolą, a następnie niejawne otwarcie tego pliku.

● Ponieważ wykonanie procedury *Close* dotyczącej pliku skojarzonego z urządzeniem logicznym nie ma żadnych skutków, zamknięcie rozpatrywanego pliku tekstowego byłoby możliwe dopiero po wykonaniu innej procedury *Assign* dotyczącej zmiennej *Console*, np.

```
Assign(Console,'KBD:');
```

■

W celu ułatwienia posługiwania się plikami skojarzonymi z urządzeniami logicznymi wprowadzono w Turbo Pascalu wiele predefiniowanych zmiennych plikowych, reprezentujących pliki tekstowe i dokonano niejawnych skojarzeń tych plików z wybranymi urządzeniami logicznymi.

Tablica 12.1. Skojarzenia plików tekstowych i urządzeń logicznych

Zmienna plikowa	Urządzenie logiczne
Input	CON: albo TRM:
Output	CON: albo TRM:
Con	CON:
Trm	TRM:
Kbd	KBD:
Lst	LST

Jak wynika z tablicy 12.1 z predefiniowanymi nazwami zmiennych plikowych są związane ustalone urządzenia logiczne. Wyjątek od tej zasady dotyczy zmiennych plikowych *Input* i *Output*, z których każda może reprezentować plik skojarzony z urządzeniem CON: albo z urządzeniem TRM:.

Wybór CON: albo TRM: odbywa się na podstawie dyrektywy kompilatora {$B+} albo {$B-}. Przez domniemanie przyjmuje się {$B+}, kiedy to plik reprezentowany przez *Input* jak i przez *Output* jest kojarzony z urządzeniem CON:. W zasięgu dyrektywy {$B-} odbywa się natomiast skojarzenie z urządzeniem TRM:.

Należy zwrócić uwagę, że pliki reprezentowane przez wymienione zmienne plikowe są zawsze otwarte, a operacje na nich dotyczą ustalonych urządzeń logicznych.

Przykład

```
begin
  Writeln(Con,'Hello world')
end
```

• *Con* jest predefiniowaną zmienną plikową, reprezentującą plik skojarzony z konsolą.

• Wykonanie przytoczonego programu powoduje wyprowadzenie na konsolę napisu *Hello world.* ■

12.3. OPERACJE NA PLIKACH TEKSTOWYCH

Jak już wyjaśniono, pliki tekstowe składają się z wierszy podzielonych na znaki. Każdy wiersz jest zakończony parą znaków CR/LF, a ostatnim znakiem pliku jest Ctrl-Z. Znak ten jest umieszczany w pliku otwartym do wyprowadzania lub rozszerzania - w chwili zamykania pliku.

Pliki tekstowe mogą być kojarzone ze zbiorami danych albo z urządzeniami logicznymi. W pierwszym przypadku przetwarzanie danych zawartych w pliku musi być poprzedzone wykonaniem procedury *Assign*, a następnie procedury *Reset* albo *Rewrite*. Na zakończenie tego przetwarzania powinna być wykonana procedura *Close*. W drugim przypadku, tj. skojarzenia pliku z urządzeniem, można posłużyć się predefiniowaną zmienną plikową. W takim przypadku zabrania się wykonania procedur *Assign*, *Reset*, *Rewrite*, *Close* dotyczących tej zmiennej, albo też można potraktować urządzenie tak jakby było zbiorem danych o nazwie symbolicznej urządzenia, posłużyć się procedurą *Assign* dla skojarzenia pliku z urządzeniem, a następnie przystąpić do przetwarzania pliku.

Procedura Assign

Wywołanie: *Assign(TextVar,StrExp)*

Przyjmuje się, że *TextVar* jest nazwą zmiennej plikowej typu *text*, a *StrExp* jest wyrażeniem łańcuchowym. Wymaga się, aby plik reprezentowany przez *TextVar* nie był otwarty. Zabrania się, aby *TextVar* było nazwą predefiniowanej zmiennej plikowej.

Wykonanie procedury *Assign* powoduje skojarzenie pliku reprezentowanego przez zmienną *TextVar* ze zbiorem danych albo z urządzeniem logicznym o nazwie określonej przez wartość danej reprezentowanej przez wyrażenie *StrExp*.

Przykład

```
var
 Device : text;
...
Assign(Device,'CON:');
```

- Zmienna plikowa *Device* reprezentuje plik tekstowy.
- Wykonanie procedury *Assign* powoduje skojarzenie pliku reprezentowanego przez zmienną *Device* z konsolą.
- Wykonanie procedury *Assign* powoduje otwarcie pliku reprezentowanego przez zmienną *Device*. ■

Procedura Reset

Wywołanie: *Reset(TextVar)*

Przyjmuje się, że *TextVar* jest nazwą zmiennej plikowej typu *text*. Wymaga się, aby przed wywołaniem procedury *Reset* plik reprezentowany przez zmienną *TextVar* był skojarzony z istniejącym zbiorem danych albo z urządzeniem logicznym. Zabrania się, aby *TextVar* było nazwą predefiniowanej zmiennej plikowej.

Wykonanie procedury *Reset* powoduje otwarcie pliku reprezentowanego przez zmienną *TextVar*. Jeśli plik reprezentowany przez *TextVar* jest skojarzony z urządzeniem logicznym, to jest już otwarty i użycie procedury *Reset* nie ma żadnych skutków.

Przykład

```
var
  InpFile : text;
...
Assign(InpFile,'OLDBOOK');
Reset(InpFile);
```

- Wykonanie procedury *Assign* powoduje skojarzenie pliku tekstowego reprezentowanego przez zmienną *InpFile* ze zbiorem *OLDBOOK*.
- Wykonanie procedury *Reset* powoduje otwarcie pliku tekstowego reprezentowanego przez zmienną *InpFile* w trybie do wprowadzania. ■

Procedura Rewrite

Wywołanie: *Rewrite(TextVar)*

Przyjmuje się, że *TextVar* jest nazwą zmiennej plikowej typu *text*. Wymaga się, aby przed wywołaniem procedury *Rewrite*

plik reprezentowany przez zmienną *TextVar* był skojarzony ze zbiorem danych albo z urządzeniem logicznym. Zabrania się, aby *TextVar* było nazwą predefiniowanej zmiennej plikowej.

Wykonanie procedury *Rewrite* powoduje otwarcie pliku reprezentowanego przez zmienną *TextVar*. Jeśli plik reprezentowany przez *TextVar* jest skojarzony z urządzeniem logicznym, to jest już otwarty i użycie procedury *Rewrite* nie ma żadnych skutków. Jeśli plik reprezentowany przez *TextVar* jest skojarzony ze zbiorem danych, a zbiór ten nie istnieje, to zostanie utworzony. Jeśli istnieje, to zostanie usunięty i utworzony ponownie. W obu przypadkach zostanie utworzony zbiór pusty.

Przykład

```
var
  OutFile : text;
...
Assign(OutFile,'NEWBOOK');
Rewrite(OutFile);
```

- Wykonanie procedury *Assign* powoduje skojarzenie pliku tekstowego reprezentowanego przez zmienną *OutFile* ze zbiorem *NEWBOOK*.
- Wykonanie procedury *Rewrite* powoduje otwarcie pliku reprezentowanego przez zmienną *OutFile* w trybie do wyprowadzania. Po dokonaniu otwarcia plik reprezentowany przez tę zmienną jest pusty. ■

Procedura Read

Wywołanie: *Read(TextVar,VarList)*

Przyjmuje się, że *TextVar* jest nazwą zmiennej plikowej typu *text*, a *VarList* jest nazwą zmiennej albo listą nazw zmiennych typu *char*, *string*, *integer* lub *real*. Jeśli *TextVar* jest nazwą *Input*, to wywołanie może zostać uproszczone do *Read(VarList)*. Jeśli wywołanie z jawną albo domniemaną nazwą *Input* znajduje się w zasięgu dyrektywy {$B+}, to jest traktowane tak, jakby *TextVar* było nazwą *Con*. Jeśli znajduje się w zasięgu dyrektywy {$B-}, to jest traktowane tak, jakby było nazwą *Trm*. W każdym przypadku wymaga się, aby plik reprezentowany przez zmienną *TextVar* był otwarty.

Wykonanie procedury *Read* powoduje wprowadzenie ciągu znaków z pliku reprezentowanego przez zmienną *TextVar*, zinterpretowanie ich jako umownych zapisów wartości danych, a następnie przypisanie danych o tych wartościach zmiennym reprezentowanym przez *VarList*.

Należy nadmienić, że jeśli wprowadzanie dotyczy urzą-

dzenia logicznego CON:, to każde wykonanie procedury *Read* powoduje wprowadzanie nowego wiersza tekstu, i to nawet wówczas, gdy w buforze konsoli pozostają nie wprowadzone jeszcze znaki. W takim przypadku interpretowanie znaków bufora następuje dopiero po zakończeniu wiersza znakiem CR, a ewentualne wcześniejsze wprowadzenie znaku Ctrl-Z jest ignorowane. Znak taki jest zawsze umieszczany na końcu bufora, ale dopiero po wprowadzeniu znaku CR.

Sposób interpretowania znaków pliku reprezentowanego przez zmienną *TextVar* zależy od typu zmiennych *VarList*.

Dla zmiennych typu *char*:
Wprowadzany jest jeden znak i przypisywany zmiennej.

Dla zmiennych typu **string**:
Wprowadzany jest najdłuższy dopuszczalny ciąg znaków i przypisywany zmiennej. Liczba wprowadzonych znaków nie przekracza maksymalnego rozmiaru zmiennej łańcuchowej, ani nie jest większa niż liczba znaków bieżącego wiersza.

Dla zmiennych typu *integer*:
Wprowadzany jest ciąg znaków mający postać liczby całkowitej, po którym następuje jeden ze znaków: spacja, tabulacja, CR albo Ctrl-Z. Znaki spacji, tabulacji, CR i LF poprzedzające wspomnianą liczbę są ignorowane. Następnie dana o wartości wprowadzonej liczby zostaje przypisana kolejnej zmiennej *VarList*. Jeśli wprowadzony ciąg znaków ma inną postać niż wynika z opisu, to powstaje błąd wykonania operacji wejścia/wyjścia.

zmiennych typu *real*:
Wprowadzany jest ciąg znaków mający postać liczby rzeczywistej, po której następuje jeden ze znaków: spacja, tabulacja, CR albo Ctrl-Z. Znaki spacji, tabulacji, CR i LF poprzedzające liczbę są ignorowane. Następnie dana o wartości wprowadzonej liczby zostaje przypisana kolejnej zmiennej *VarList*. Jeśli wprowadzony ciąg znaków ma inną postać niż wynika to z opisu, to powstaje błąd wykonania operacji wejścia/wyjścia.

Przykład

```
var
  IntVar : integer;
  FixVar : real;
  TextVar : text;
  Ter1,Ter2 : char;
  Str : string[8];
begin
  Assign(TextVar,'ONELINE.DOC');
```

```
    Reset(TextVar);
    Read(TextVar,IntVar,Ter1,FixVar,Ter2,Str);
    ...
end.
```

● Jeśli przyjąć, że pierwszym wierszem tekstu zawartym w zbiorze *ONELINE.DOC* jest ciąg

12 -13.8 Izabela

to zmiennej *IntVar* zostanie przypisana dana o wartości 12, zmiennej *FixVar* dana o wartości -13.8, a zmiennej *Str* dana o wartości 'Izabela'.

● Każdej ze zmiennych *Ter1* i *Ter2* zostanie przypisana spacja. Oznacza to, że znak kończący liczbę podlega podwójnej interpretacji, jako ostatni znak zapisu liczby i jako pierwszy znak ciągu następującego po liczbie.

● Gdyby w rozpatrywanym ciągu dokonać zmiany znaku - (minus) na + (plus), to podczas wykonywania procedury *Read* powstałby błąd operacji wejścia/wyjścia. Wynika to z faktu, że +13.8 nie jest poprawnym literałem liczbowym. ■

Jeśli podczas wykonywania procedury *Read* plik znajdzie się w pozycji końcowej, a nie wszystkim zmiennym *VarList* zostały już przypisane dane, to każdej zmiennej typu *char* zostanie przypisany znak Ctrl-Z, każdej zmiennej typu *string* zostanie przypisany pusty ciąg znaków, a zmiennym typu *integer* i *real* nie zostaną przypisane żadne dane.

Przykład

```
var
  IntVar : integer;
  FixVar : real;
  TextVar : text;
  Ter1,Ter2 : char;
  Str : string[4];
begin
  IntVar := -44;
  FixVar := -44;
  StrVar := 'janb';
  Ter1 := 'x';
  Ter2 := 'y';
  Assign(TextVar,'SHORT.DOC');
  Reset(TextVar);
  Read(TextVar,IntVar,Ter1,FixVar,Ter2,Str);
  ...
end.
```

● Jeśli przyjąć, że zbiór *SHORT.DOC* składa się ze znaków: 1, 3, spacja, CR, LF, Ctrl-Z, to po wykonaniu instrukcji

Read zmienna *IntVar* ma wartość 13, zmienna *FixVar* ma wartość -44.0, zmienna *Ter1* ma wartość #32 (spacja), zmienna *Ter2* ma wartość #26 (Ctrl-Z), a zmienna *Str* ma wartość ''. ■

Należy nadmienić, że jeśli plik jest skojarzony z konsolą, to na końcu każdego wprowadzanego wiersza jest umieszczany znak Ctrl-Z. Ma to taki skutek, jakby przed końcem każdego wiersza plik znajdował się w pozycji końcowej.

Przykład

```
var
  IntVar : integer;
  StrVar : string[3];
  ChrVar : char;
begin
  IntVar := -2;
  StrVar := 'jan';
  ChrVar := 'b';
  Read(IntVar,StrVar,ChrVar);
  ...
end.
```

● Wywołanie procedury *Read* jest interpretowane tak jak wywołanie

```
Read(Input,IntVar,StrVar,ChrVar);
```

● Ponieważ wymienione wywołanie znajduje się w zasięgu domniemanej dyrektywy {$B+}, więc jest interpretowane tak jak wywołanie

```
Read(Con,IntVar,StrVar,ChrVar);
```

● Jeśli podczas wykonywania procedury *Read* zostanie wprowadzony z konsoli tylko jeden znak CR, to spowoduje to zakończenie wykonywania tej instrukcji. W tym momencie zmienna *IntVar* będzie miała wartość -2, zmienna *StrVar* - wartość '', a zmienna *ChrVar* - wartość #26 (Ctrl-Z). ■

Procedura Readln

Wywołanie: *Readln(TextVar)*
Readln(TextVar,VarList)

Przyjmuje się, że *TextVar* jest nazwą zmiennej plikowej typu *text*, a *VarList* jest nazwą zmiennej albo listą nazw zmiennych typu *char*, *string*, *integer* lub *real*. Jeśli *TextVar* jest nazwą *Input*, to wywołanie w pierwszej postaci może być uproszczone do *Readln*, a w drugiej do *Readln(VarList)*.

Wywołanie procedury *Readln* w postaci *Readln(TextVar)* powoduje wprowadzenie i zignorowanie ciągu znaków pliku aż do CR/LF włącznie. Jeśli plik jest skojarzony z urządzeniem lo-

gicznym, to są pomijane znaki tylko do CR włącznie. Jeśli *TextVar* reprezentuje konsolę, to na ekran monitora zostaną wyprowadzone znaki CR/LF. Natomiast wywołanie tej procedury w postaci

Readln(TextVar,VarList)

jest równoważne parze wywołań

Read(TextVar,VarList); Readln(TextVar)

i z tego powodu nie wymaga dodatkowych wyjaśnień.

Przykład

```
var
  Count : integer;
  Source : text;
  Line : string[128];
begin
  Assign(Source,'VOLUME.DOC');
  Reset(Source);
  Count := 0;
  while not Eof(Source) do begin
     Readln(Source,Line);
     Count := Count + 1
  end;
  Writeln(Count)
end.
```

● Wykonanie przytoczonego programu powoduje wyznaczenie liczby wierszy zbioru danych *VOLUME.DOC*. ■

Procedura Write

Wywołanie: *Write(TextVar,ExpList)*

Przyjmuje się, że *TextVar* jest nazwą zmiennej plikowej typu *text*, a *ExpList* jest wyrażeniem albo listą wyrażeń typu *char*, *string*, *integer*, *real* lub *boolean*. Bezpośrednio po każdym z wymienionych wyrażeń może wystąpić kwalifikator o postaci *:m*, a po wyrażeniu typu *real* także kwalifikator o postaci *:m:n*, w którym *m* i *n* są wyrażeniami typu *integer*. Jeśli *TextVar* jest nazwą *Output*, to wywołanie może zostać uproszczone do *Write(ExpList)*. Wywołanie z jawną albo domniemaną nazwą *Output* jest traktowane tak, jakby *TextVar* było nazwą *Con* (lub równoważną jej w danym kontekście nazwą *Trm*). W każdym przypadku wymaga się, aby plik reprezentowany przez zmienną *TextVar* był otwarty.

Wykonanie procedury *Write* powoduje wyprowadzenie do pliku reprezentowanego przez zmienną *TextVar* ciągu znaków składającego się z umownych zapisów wartości danych reprezentowanych przez wyrażenia *ExpList*.

Sposób tworzenia znaków wspomnianego ciągu zależy od typu wyrażeń *ExpList* oraz od wartości kwalifikatorów *m* i *n*.

Dla wyrażeń typu *char*:
Wyprowadzany jest znak reprezentowany przez wyrażenie. Jeśli posłużono się kwalifikatorem *m*, a *m*>1, to znak ten zostanie poprzedzony *m*−1 spacjami.

Dla wyrażeń typu **string**:
Wyprowadzany jest ciąg znaków reprezentowany przez wyrażenie. Jeśli posłużono się kwalifikatorem *m*, a *m* jest większe niż liczba *l* znaków tego ciągu, to ciąg ten zostanie poprzedzony *m*−*l* spacjami.

Dla wyrażeń typu *integer*:
Wyprowadzany jest ciąg znaków o postaci najkrótszego literału reprezentującego wartość wyrażenia. Jeśli posłużono się kwalifikatorem *m*, a *m* jest większe niż liczba *l* znaków wspomnianego literału, to ciąg ten zostanie poprzedzony *m*−*l* spacjami.

Dla wyrażeń typu *real*:
Wyprowadzany jest ciąg znaków o postaci *bsd.ddddddddddEsdd* reprezentujący wartość wyrażenia. W ciągu tym *b* jest spacją, a pierwsze *s* jest znakiem przedstawianym jako spacja dla wyrażenia o wartości nieujemnej, a jako znak − (minus) dla wyrażenia o wartości ujemnej. Każde *d* reprezentuje w tym ciągu cyfrę dziesiętną, a drugie *s* jest znakiem + (plus) albo − (minus). Jeśli posłużono się kwalifikatorem *m*, a *m*>18, to przedstawiony ciąg znaków zostanie dodatkowo poprzedzony *m*−18 spacjami. Jeśli *m*=18, to zostanie wyprowadzony ciąg wzorcowy podany na początku opisu, jeśli natomiast *m*<18, to wyprowadzany ciąg zostanie skrócony do *m* znaków, pochodzących z ciągu wzorcowego przez odrzucenie z niego w pierwszej kolejności spacji wiodących, a następnie końcowych cyfr mantysy, ale z zaokrągleniem. Jeśli wartość *m* jest taka, że wymagałoby to pozostawienia mniej niż 2 cyfr mantysy, to poprzestaje się na odrzuceniu cyfry trzeciej i następnych, także z zaokrągleniem. Jeśli posłużono się kwalifikatorem o postaci :*m*:*n*, to jest wyprowadzany ciąg znaków mający postać literału rzeczywistego bez wykładnika, z *n* cyframi ułamkowymi, wyrównanego prawostronnie w ciągu o długości *m* znaków. Jeśli wspomniany literał nie da się przedstawić za pomocą ciągu o tej liczbie znaków, to *m* zostaje niejawnie zwiększone o tyle, aby było to wykonalne. Należy przy tym uwzględnić, że jeśli *n*=0, to literał jest wyprowadzany również bez części ułamkowej i kropki, a jeśli *n* nie należy do przedziału domkniętego [0,24], to kwalifikator :*m*:*n* jest traktowany tak jak uprzednio opisany kwalifikator :*m*.

Dla wyrażeń typu *boolean*:
Wyprowadzany jest ciąg znaków o postaci *TRUE* albo *FALSE*, reprezentujący wartość wyrażenia. Jeśli posłużono się kwalifikatorem *m*, a *m* jest większe niż liczba *l* znaków tego ciągu, to ciąg zostanie poprzedzony *m*−*l* spacjami.

Przykłady (symbol *b* oznacza spację)

Wyrażenie	Ciąg wyprowadzany
'j'	*j*
'j':2	*bj*
'jan'	*jan*
'jan':4	*bjan*
44	44
−44	−44
44:4	*bb*44
23.5	*bb*2.3500000000E+01
23.456789:10	2.3457E+01
23.456789:6	2.3E+01
23.456789:6:2	*b*23.46
−23.456789:6:0	*bbb*−23
23.456789:6:−2	2.3E+01

● Jak wynika z przykładów, zaokrąglenie jest dokonywane na podstawie pierwszej cyfry odrzucanej. ■

Procedura Writeln

Wywołanie: *Writeln*(*TextVar*)
Writeln(*TextVar*,*ExpList*)

Przyjmuje się, że *TextVar* jest nazwą zmiennej plikowej typu *text*, a *ExpList* jest nazwą zmiennej albo listą nazw zmiennych typu *char*, *string*, *integer*, *real* lub *boolean*. Bezpośrednio po każdym z wymienionych wyrażeń może wystąpić kwalifikator o postaci :*m*, a po wyrażeniu typu *real* także kwalifikator o postaci :*m*:*n*, gdzie *m* i *n* są wyrażeniami typu *integer*. Jeśli *TextVar* jest nazwą *Output*, to wywołanie w pierwszej postaci może być uproszczone do *Writeln*, a w drugiej do *Writeln*(*ExpList*).

Wywołanie procedury *Writeln* o postaci *Writeln*(*TextVar*) powoduje wyprowadzenie do pliku reprezentowanego przez zmienną *TextVar* pary znaków CR/LF. Natomiast wywołanie tej procedury w postaci

Writeln(*TextVar*,*ExpList*)

jest równoważne parze wywołań

Write(*TextVar*,*ExpList*); *Writeln*(*TextVar*);

i z tego powodu nie wymaga dodatkowych wyjaśnień.

Przykład

```
begin
   Writeln(false);
   Writeln;
   Writeln(true)
end.
```

● Wykonanie przytoczonego programu powoduje wyprowadzenie na konsolę 3 wierszy tekstu.

● Drugi z wyprowadzonych wierszy jest pusty.

● Ogółem zostanie wyprowadzonych 15 znaków, w tym napisy *FALSE* i *TRUE* oraz dwie pary znaków CR/LF. ■

Procedura Close

Wywołanie: *Close(TextVar)*

Przyjmuje się, że *TextVar* jest nazwą zmiennej plikowej typu *text*. Zabrania się, aby *TextVar* było nazwą predefiniowanej zmiennej plikowej.

Wywołanie procedury *Close* powoduje zamknięcie pliku reprezentowanego przez zmienną plikową *TextVar*.

Przykład

```
var
  Output : text;
begin
  Assign(Output,'JB.DOC');
  Rewrite(Output);
  Writeln(Output,'jasio');
  Close(Output)
end.
```

● Wobec jawnej deklaracji nazwa *Output* nie jest nazwą predefiniowanej zmiennej plikowej, więc wywołanie procedury *Assign* jest poprawne i niezbędne.

● Wykonanie przytoczonego programu powoduje utworzenie zbioru zawierającego 8 znaków.

● Wywołanie procedury *Close* powoduje wyprowadzenie znaku Ctrl-Z oraz wykonanie czynności kończących utworzenie zbioru *JB.DOC* zawierającego ciąg znaków: jasio CR LF Ctrl-Z. ■

Funkcja Eof

Wywołanie: *Eof(TextVar)*

Przyjmuje się, że *TextVar* jest nazwą zmiennej plikowej

typu *text*. Jeśli *TextVar* jest nazwą *Input*, to wywołanie może być uproszczone do *Eof*.

Wywołanie funkcji *Eof* może dotyczyć zarówno pliku skojarzonego ze zbiorem danych, jak i pliku skojarzonego z urządzeniem logicznym. W obu przypadkach rezultatem funkcji jest dana typu *boolean* o wartości *true* albo *false*.

Jeśli plik jest skojarzony ze zbiorem danych, to rezultatem funkcji *Eof* jest dana o wartości *true*, o ile plik znajduje się w pozycji przed znakiem Ctrl-Z albo w pozycji końcowej. W przeciwnym razie rezultatem tej funkcji jest dana o wartości *false*.

Jeśli plik jest skojarzony z urządzeniem logicznym, to rezultatem funkcji *Eof* jest dana o wartości *true* - jeśli ostatnim zinterpretowanym znakiem był Ctrl-Z. W przeciwnym razie rezultatem tej funkcji jest dana o wartości *false*.

Przykład

```
var
  ChrVar : char;
begin
  Read(ChrVar);
  Writeln(Eof)
end.
```

● Jeśli podczas wykonywania przytoczonego programu zostaną wprowadzone z konsoli znaki Ctrl-Z i CR, to nastąpi wyprowadzenie napisu *FALSE*.

● Stanie się tak dlatego, że znak Ctrl-Z zostanie pominięty, a więc rezultatem funkcji *Eof* będzie dana o wartości *false*. ■

Funkcja Eoln

Wywołanie: *Eoln(TextVar)*

Przyjmuje się, że *TextVar* jest nazwą zmiennej plikowej typu *text*. Jeśli *TextVar* jest nazwą *Input*, to wywołanie może zostać uproszczone do *Eoln*.

Wywołanie funkcji *Eoln* może dotyczyć zarówno pliku skojarzonego ze zbiorem danych, jak i pliku skojarzonego z urządzeniem logicznym. W obu przypadkach rezultatem funkcji jest dana typu *boolean* o wartości *true* albo *false*.

Jeśli plik jest skojarzony ze zbiorem danych, to rezultatem funkcji *Eoln* jest dana o wartości *true* - jeśli plik znajduje się w pozycji przed znakiem CR albo gdy rezultatem funkcji *Eof* jest dana o wartości *true*. W przeciwnym razie rezultatem funkcji *Eoln* jest dana o wartości *false*.

Jeśli plik jest skojarzony z urządzeniem logicznym, to

rezultatem funkcji *Eoln* jest dana o wartości *true* - jeśli ostatnim zinterpretowanym znakiem był CR albo gdy rezultatem funkcji *Eof* jest dana o wartości *true*. W przeciwnym razie rezultatem funkcji *Eoln* jest dana o wartości *false*.

Przykład

```
var
  ChrVar : char;
  TxtVar : text;
begin
  Assign(TxtVar,'STRANGE.DOC');
  Reset(TxtVar);
  while not Eof(TxtVar) do begin
     Writeln(Eoln(TxtVar));
     Read(TxtVar,ChrVar)
  end
end.
```

● Jeśli zbiór *STRANGE.DOC* składa się z jednego wiersza pustego, tj. ze znaków CR, LF, Ctrl-Z, to wykonanie przytoczonego programu powoduje wyprowadzenie napisu

TRUE
FALSE

● Wynika stąd, że w pozycji przed znakiem LF rezultatem funkcji *Eoln* jest dana o wartości *false*. ■

Funkcja SeekEof

Wywołanie: *SeekEof(TextVar)*

Przyjmuje się, że *TextVar* jest nazwą zmiennej plikowej typu *text*. Jeśli *TextVar* jest nazwą *Input*, to wywołanie może zostać uproszczone do *SeekEof*.

Wywołanie funkcji *SeekEof* jest zbliżone do wywołania funkcji *Eof*. Bezpośrednio po pominięciu najbliższych spacji, tabulacji oraz znaków CR i LF jest udostępniany taki sam rezultat jak dla funkcji *Eof*.

Przykład

```
var
  TxtVar : text;
begin
  Assign(TxtVar,'STRANGE.DOC');
  Reset(TxtVar);
  Writeln(SeekEof(TxtVar))
end.
```

● Jeśli zbiór *STRANGE.DOC* składa się z jednego wiersza pustego, to wykonanie przytoczonego programu powoduje wypro-

wadzenie napisu *TRUE*. ■

Funkcja SeekEoln

Wywołanie: *SeekEoln(TextVar)*

Przyjmuje się, że *TextVar* jest nazwą zmiennej plikowej typu *text*. Jeśli *TextVar* jest nazwą *Input*, to wywołanie może zostać uproszczone do *SeekEoln*.

Wykonanie funkcji *SeekEoln* jest zbliżone do wykonania funkcji *Eoln*. Bezpośrednio po pominięciu najbliższych spacji i tabulacji jest udostępniany taki sam rezultat jak dla funkcji *Eoln*.

Przykład

```
var
  TxtVar : text;
  Tally : integer;
  ChrVar : char;
begin
  Tally := 0;
  Assign(TxtVar,'TEXT.DOC');
  Reset(TxtVar);
  while not SeekEof(TxtVar) do begin
    while not SeekEoln(TxtVar) do begin
      Tally := Tally + 1;
      Read(TxtVar,ChrVar);
    end
  end;
  Writeln(Tally)
end.
```

● Wykonanie przytoczonego programu powoduje wyznaczenie liczby znaków zbioru *TEXT.DOC*.

● Spacje, tabulacje, znaki CR, LF i Ctrl-Z kończące wiersze są ignorowane. ■

12.4. PLIKI BEZ TYPU ELEMENTOW

Pliki bez typu elementów umożliwiają wykonywanie niebuforowanych operacji wprowadzania/wyprowadzania, dokonywanych bezpośrednio między zmiennymi programu a zewnętrzną pamięcią dyskową. Przyjmuje się, że elementami pliku bez typu elementów są obiekty zajmujące po 128 bajtów pamięci. Plik bez typu elementów może reprezentować dowolny zbiór dyskowy. Z tego względu operacje takie jak np. *Erase* lub *Rename* mogą być wykonywane za pośrednictwem plików bez typu elementów.

Opis typu pliku bez typu elementów składa się ze słowa kluczowego **file**.

Składnia

opis-typu-plikowego-bez-typu-elementów:
file

Przykład

```
var
  DiskFile : file;
  FileName : string[40];
begin
  Write('FileName :- ');
  Readln(FileName);
  Assign(DiskFile,FileName);
  Erase(DiskFile)
end.
```

- Wykonanie przytoczonego programu powoduje usunięcie dowolnego zbioru dyskowego o nazwie wprowadzonej z klawiatury konsoli.
- Typ elementów zbioru jest bez znaczenia. ■

Wykonywanie operacji wprowadzania/wyprowadzania na plikach bez typu elementów jest zrealizowane za pomocą procedur *BlockRead* i *BlockWrite*. Zastępują one odpowiednio procedury z typem elementów. Pozostałe operacje jak *Assign*, *Reset*, *Rewrite*, *Close*, *Seek* i *Eof* mają taką samą interpretację jak dla plików z ustalonym typem elementów.

Procedura BlockRead

Wywołanie: *BlockRead(FileVar,Buffer,Count,Reply)*
BlockRead(FileVar,Buffer,Count)

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej reprezentującej plik bez typu elementów, *Buffer* jest nazwą dowolnej zmiennej programu, *Count* jest wyrażeniem typu *integer*, a *Reply* jest nazwą zmiennej typu *integer*.

Wykonanie procedury *BlockRead* powoduje wprowadzenie z pliku reprezentowanego przez zmienną *FileVar*, do obszaru pamięci operacyjnej zajmowanego przez zmienną *Buffer*, *Count* obiektów o rozmiarze 128 bajtów każdy. Jeśli posłużono się wywołaniem zawierającym argument *Reply*, to zmiennej reprezentowanej przez ten argument zostanie przypisana dana określająca liczbę faktycznie wprowadzonych obiektów. Jeśli liczba ta jest mniejsza niż *Count*, to oznacza to, że plik znajduje się w pozycji końcowej.

Procedura BlockWrite

Wywołanie: *BlockWrite(FileVar,Buffer,Count,Reply)*
BlockWrite(FileVar,Buffer,Count)

Przyjmuje się, że *FileVar* jest nazwą zmiennej plikowej reprezentującej plik bez typu elementów, *Buffer* jest nazwą dowolnej zmiennej programu, *Count* jest wyrażeniem typu *integer*, a *Reply* jest nazwą zmiennej typu *integer*.

Wykonanie procedury *BlockWrite* powoduje wyprowadzenie do pliku reprezentowanego przez zmienną *FileVar*, z obszaru pamięci zajmowanego przez zmienną *Buffer*, *Count* obiektów o rozmiarze 128 bajtów każdy. Jeśli posłużono się wywołaniem zawierającym *Reply*, to zmiennej reprezentowanej przez ten argument zostanie przypisana dana określająca liczbę faktycznie wyprowadzonych obiektów. Jeśli liczba ta jest mniejsza niż *Count*, to oznacza to, że wykonanie procedury nie przebiegło pomyślnie.

Przykład

```
program Copy;
var
  Src,Trg : file;
  Buffer : array[0..255,boolean] of byte;
  Source,Target : string[40];
  Reply : integer;
begin
  Write('Source :- ');
  Readln(Source);
  Write('Target :- ');
  Readln(Target);
  Assign(Src,Source);
  Assign(Trg,Target);
  Reset(Src);
  Rewrite(Trg);
  repeat
    BlockRead(Src,Buffer,4,Reply);
    BlockWrite(Trg,Buffer,Reply)
  until Reply < 4;
  Close(Src);
  Close(Trg)
end.
```

● Wykonanie przytoczonego programu powoduje skopiowanie zbioru danych o dowolnym typie elementów. ■

12.5. KONTROLOWANIE POPRAWNOSCI OPERACJI WEJSCIA/WYJSCIA

Sposób kontrolowania poprawności operacji wejścia/wyjścia jest uzależniony od sposobu skompilowania programu. W zakresie przyjmowanej przez domniemanie dyrektywy {$I+}, po każdej operacji wejścia/wyjścia jest kontrolowana jej poprawność. Jeśli zostanie wykryty błąd, to wykonywanie programu jest przerywane, a na konsolę jest wyprowadzany komunikat o rozpoznanym błędzie. W zakresie dyrektywy {$I-} rozpoznanie błędu nie powoduje wstrzymania wykonywania programu, a jedynie wstrzymanie wykonywania dalszych operacji wejścia/wyjścia. Stan taki utrzymuje się aż do momentu wywołania funkcji standardowej *IOresult*, udostępniającej daną typu *integer*. Jeśli rezultatem wywołania tej funkcji jest dana o wartości 0, to oznacza to, że dotychczasowe operacje wejścia/wyjścia zostały wykonane poprawnie. Jeśli rezultatem jest dana o wartości różnej od zera, to wartość ta określa rodzaj rozpoznanego błędu. Wykaz błędów operacji wejścia/wyjścia przytoczono w dodatku C.

Przykład

```
program Delete;
var
  FileVar : file;
  FileName : string[40];
  Flag : boolean;
begin
  Write('FileName :- ');
  Readln(FileName);
  Assign(FileVar,FileName);
  {$I-} Erase(FileVar); {$I+}
  if not (IOresult = 0) then
    Writeln('File ' + FileName + 'did not exist')
end.
```

- Wykonanie przytoczonego programu powoduje usunięcie zbioru danych o nazwie wprowadzonej z konsoli.
- Jeśli podano nazwę zbioru nie istniejącego, to zostanie wyprowadzony komunikat, że zbiór ten nie istniał.
- Gdyby z programu usunięto dyrektywy kompilatora, a zbiór by nie istniał, to wykonywanie programu zostałoby przerwane na skutek wystąpienia błędu operacji *Erase* dotyczącej zbioru, który nie istnieje. ■

Każdy z typów wskazujących jest typem prostym. Wartościami danych wskazujących są wskazania danych.

Opis typu wskazującego składa się ze znaku ^ (caret), bezpośrednio po którym występuje nazwa typu. Nie wymaga się, aby nazwa ta była już zdefiniowana. Jest to jedyny wyjątek od wymagania powoływania się na obiekty uprzednio zdefiniowane.

Składnia

Opis-typu-wskazującego:
nazwa-typu
nazwa-typu:
identyfikator

Przykład

```
type
  EmployeePtr = ^EmployeeData;
  EmployeeData = record
                   Name : string[30];
                   Salary : real
                 end;
var
  EmployeeRef : EmployeePtr;
  IntRef : ^integer;
```

- Typ *EmployeePtr* jest związany ze zbiorem wskazań danych typu *EmployeeData*.
- *EmployeeRef* jest nazwą zmiennej wskazującej. Zmiennej tej mogą być przypisywane dane wskazujące dane typu *EmployeeData*.
- *IntRef* jest nazwą zmiennej wskazującej. Zmiennej tej mogą być przypisywane dane wskazujące dane typu *integer*. ■

Poza typami i zmiennymi wskazującymi występuje w Turbo Pascalu wskazanie puste reprezentowane przez słowo kluczowe **nil**. Typ tego wskazania jest zgodny z dowolnym innym typem wskazującym. Zmienne wskazujące oraz wskazanie puste mogą występować w relacjach = (równe) i <> (nie równe). W instrukcji przypisania wymaga się zgodności typów lewej i prawej strony. Zgodność ta ma miejsce tylko wtedy, gdy prawa strona jest słowem kluczowym **nil** oraz wtedy, gdy prawa strona reprezentuje daną wskazującą takiego samego typu jak dane wskazywane przez zmienną znajdującą się z lewej strony symbolu przypisania.

Przykład

```
type
  Days = (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
  Week = Mon..Sun;
var
  Day1 : ^Days;
  Day2,Day3 : ^Week;
```

● W zasięgu przytoczonych definicji i deklaracji poprawne są m.in. przypisania

```
Day1 := nil;
Day2 := Day3
```

ale nie jest poprawne przypisanie

```
Day2 := Day1;
```

ponieważ zmienne *Day2* i *Day1* są różnych typów. ■

Zmienne wskazujące mogą zostać wykorzystane do dynamicznego zarządzania pamięcią operacyjną. Pomocne są w tym procedury do przydzielania i zwalniania tej pamięci.

Procedura New

Wywołanie: *New(PtrVar)*

Przyjmuje się, że *PtrVar* jest nazwą zmiennej wskazującej.

Wykonanie procedury *New* powoduje utworzenie zmiennej takiego samego typu, jakiego są dane, na które mogą wskazywać dane przypisywane zmiennej *PtrVar*. Po wykonaniu tej czynności zmiennej *PtrVar* zostaje przypisana dana wskazująca właśnie utworzoną zmienną. Wymieniona tu zmienna zostaje utworzona w obszarze pamięci operacyjnej nazywanym *stertą* i może być traktowana tak, jak dowolna inna zmienna jej typu.

Przykład

```
type
   Matrix = array[boolean] of array[boolean] of real;
var
   MatrixPtr : ^Matrix;
begin
   New(MatrixPtr);
   MatrixPtr^[true][true] := 23.5;
   ...
end.
```

● Wykonanie procedury *New* powoduje utworzenie – na stercie – zmiennej typu *Matrix*, a następnie przypisanie zmiennej wskazującej *MatrixPtr* danej wskazującej tę zmienną.

● Wykonanie instrukcji przypisania powoduje przypisanie narożnikowemu elementowi tablicy *MatrixPtr^* danej o wartości 23.5.

● Należy zauważyć, że napis *MatrixPtr* jest nazwą zmiennej wskazującej, a napis *MatrixPtr^* jest nazwą obiektu wskazywanego przez tę zmienną. ■

Procedura Dispose

Wywołanie: *Dispose(PtrVar)*

Przyjmuje się, że *PtrVar* jest nazwą zmiennej wskazującej.

Wykonanie procedury *Dispose* powoduje eliminację zmiennej wskazywanej przez zmienną wskazującą *PtrVar*. Wymaga się, aby eliminowana zmienna była uprzednio utworzona za pomocą procedury *New*. Obszar pamięci operacyjnej, zajmowany przez eliminowaną zmienną, zostaje zwrócony do sterty i może zostać wykorzystany podczas tworzenia innych zmiennych.

Przykład

```
var
  IntRef : ^integer;
begin
  New(IntRef);
  IntRef^ := 20;
  Dispose(IntRef);
  New(IntRef);
  IntRef^ := 30;
  Writeln(IntRef^)
end.
```

● Wykonanie przytoczonego programu powoduje wyprowadzenie napisu 30.

● Gdyby z programu usunięto drugą instrukcję przypisa-

nia, to byłby on błędny, ponieważ w momencie wykonywania procedury *Writeln* zmiennej *IntRef* nie byłaby przypisana żadna dana. ■

Procedura Mark

Wywołanie: *Mark(PtrVar)*

Przyjmuje się, że *PtrVar* jest nazwą zmiennej wskazującej dowolnego typu.

Wykonanie procedury *Mark* powoduje przypisanie zmiennej *PtrVar* wskazania bieżącego szczytu sterty.

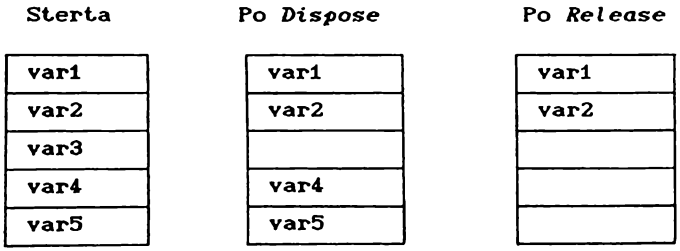
Procedura Release

Wywołanie: *Release(PtrVar)*

Przyjmuje się, że *PtrVar* jest nazwą zmiennej wskazującej dowolnego typu.

Wykonanie procedury *Release* powoduje usunięcie ze sterty zmiennej wskazywanej przez *PtrVar* i wszystkich zmiennych następujących po niej na stercie.

Procedury *New-Dispose* oraz *Mark-Release* dostarczają dwóch mechanizmów zarządzania pamięcią sterty. W ustalonym programie należy ograniczyć się do jednego z tych mechanizmów. Istotę różnic między rozpatrywanymi tu mechanizmami przedstawiono na rys.13.1. Przy założeniu, że zmiennej *Ptr* przypisano wskazanie zmiennej *var3*, pokazano tam skutek wykonania procedury *Dispose(Ptr)* i *Release(Ptr)*.



Rys.13.1. Porównanie mechanizmów zarządzania pamięcią sterty

Procedura GetMem

Wywołanie: *GetMem(PtrVar,IntExp)*

Przyjmuje się, że *PtrVar* jest nazwą zmiennej wskazującej, a *IntExp* jest wyrażeniem typu *integer*.

Wykonanie procedury *GetMem* powoduje zarezerwowanie - na stercie - obszaru pamięci operacyjnej o rozmiarze wyrażonym w bajtach i określonym przez wartość wyrażenia *IntExp*, a następnie przypisanie zmiennej *PtrVar* danej wskazującej tak przypisany obszar.

Procedura FreeMem

Wywołanie: *FreeMem(PtrVar,IntExp)*

Przyjmuje się, że *PtrVar* jest nazwą zmiennej wskazującej, a *IntExp* jest wyrażeniem typu *integer*.

Wykonanie procedury *FreeMem* powoduje zwrócenie do sterty obszaru pamięci operacyjnej wskazywanego przez zmienną *PtrVar*, zajmującego tyle bajtów, ile określa wartość danej reprezentowanej przez wyrażenie *IntExp*. Wymaga się, aby obszar zwracany za pomocą procedury *FreeMem* miał dokładnie taki sam rozmiar, jak obszar uzyskany za pomocą procedury *GetMem*.

Funkcja MaxAvail

Wywołanie: *MaxAvail*

Rezultatem funkcji *MaxAvail* jest dana typu *integer* określająca rozmiar największego spójnego obszaru pamięci operacyjnej dostępnego na stercie. Rozmiar ten jest wyrażony liczbą bajtów (Elwro 800 Jr) albo liczbą *paragrafów* po 16 bajtów (IBM PC). Jeśli rezultat jest ujemny, to faktyczną liczbę paragrafów można uzyskać dodając do rezultatu liczbę rzeczywistą 65536.

Przykład

```
type
   Pointer = ^Pair;
   Pair = record
            Int : integer;
            Ref : Pointer;
          end;
var
  Number : integer;
  Head,Tail : Pointer;
begin {$B-}
  Head := nil;
  Read(Number);
  while not Eof do begin
    New(Tail);
    Tail^.Int := Number;
    Tail^.Ref := Head;
```

```
        Head := Tail;
        Read(Number)
      end;
      Tail := Head;
      while Tail <> nil do begin
        Writeln(Tail^.Int);
        Tail := Tail^.Ref
      end
    end.
```

● Wykonanie przytoczonego programu powoduje wprowadzenie z konsoli ciągu liczb całkowitych, a następnie wyprowadzenie go w kolejności odwrotnej. ■

Jedną z ważnych właściwości języka Turbo Pascal jest możliwość *przypisywania zmiennym danych początkowych*. Jeśli podczas wykonywania programu zmienne takie zachowują swoje początkowe wartości, to mogą być traktowane jak stałe. Jeśli natomiast zmienna, której przypisano daną początkową zmieni swoją wartość, to w chwili ponownego aktywowania programu znajdującego się w pamięci operacyjnej, zostanie zachowana ostatnia wartość zmiennej z poprzedniego wykonania. Ponieważ taki program z reguły traci właściwość powtarzalności, zaleca się, aby przypisywanie danych początkowych było ograniczone do zmiennych zachowujących się jak stałe. Z tej to właśnie przyczyny deklaracje zmiennych, którym należy przypisać dane początkowe, występują w programie po słowie kluczowym **const**, a nie po słowie **var**.

Przykład

```
program Increment;
const
  Number : integer = 1;
begin
  Writeln('Number = ',Number);
  Number := Number + 1
end.
```

● W przytoczonym programie zmiennej *Number* typu *integer* przypisano daną początkową o wartości 1.

● Kolejne aktywowanie programu za pomocą dyrektywy R będzie powodować wyprowadzanie kolejnych liczb naturalnych.

■

Jak już częściowo wynika z przytoczonego przykładu, deklaracja zmiennej, której przypisano daną początkową, składa

się z nazwy zmiennej, po której następuje dwukropek, następnie opis typu zmiennej, znak równości, inicjator oraz średnik. Zapis inicjatora zależy od typu zmiennej. Dla zmiennych prostych ma on postać literału, natomiast dla zmiennych tablicowych ma postać listy inicjatorów ujętych w nawiasy okrągłe, a dla zmiennych rekordowych postać wykazu inicjatorów, także ujętego w nawiasy okrągłe.

Składnia

deklaracja-z-inicjacją:
  nazwa-zmiennej : *opis-typu* = *inicjator* ;
nazwa-zmiennej:
  identyfikator
inicjator:
  literał
  nazwa-literału
  (*lista-inicjatorów*)
  (*wykaz-inicjatorów*)

Przykłady

```
const
   LineLength : byte = 132;
   Radius : real = 13.64;
   FullName : string[12] = 'Jan Bielecki';
   CtrlM : char = ^M;
   Separator : set of char = ['/',';','.'];
   SexIsMale : boolean = true;
   ShortName : string[3] = 'Izabela';
   WholeReal : real = 44;
```

- Zmiennej *LineLength* (typu *byte*) przypisano daną początkową 132.
- Zmiennej *Radius* (typu *real*) przypisano daną początkową 13.64.
- Zmiennej *FullName* (typu **string**[12]) przypisano daną *'Jan Bielecki'*.
- Zmiennej *CtrlM* (typu *char*) przypisano daną ^*M*.
- Zmiennej *Separator* (typu **set of** *char*) przypisano daną ['/',';','.'].
- Zmiennej *SexIsMale* (typu *boolean*) przypisano daną *true*.
- Zmiennej *ShortName* (typu **string**[3]) przypisano daną *'Iza'*.
- Zmiennej *WholeReal* przypisano daną 44.0. ■

Jeśli inicjowana zmienna jest tablicą, to inicjator musi mieć postać listy ujętej w nawiasy okrągłe, wyszczególniającej dane początkowe dla wszystkich elementów tej tabli-

cy. Jeśli inicjowana zmienna jest rekordem, to inicjator musi mieć postać wykazu ujętego w nawiasy okrągłe, wyszczególniającego dane początkowe dla wszystkich albo tylko niektórych komponentów rekordu. Wymaga się, aby elementy wykazu wyszczególniały dane początkowe w takiej kolejności, w jakiej komponenty rekordu występują w jego deklaracji. W przypadku inicjowania rekordów elementami wykazu są oddzielone średnikami napisy składające się z nazwy pola rekordu, po której następuje dwukropek i inicjator pola rekordu.

Przykłady

```
type
  Color = (Red,Black,Fair);
  Person = record
             FirstName,LastName : string[12];
             Hair : Color;
             Age : byte
           end;
  Sex = (male,female);
const
  Vector : array[1..3] of Sex =
           (male,male,female);
  Matrix : array[Color,2..3] of byte =
           ((0,0),(1,1),(2,5));
  ThreeD : array[boolean,boolean,boolean] of Color =
           (((Red,Red),(Red,Fair)),
            ((Black,Black),(Red,Black)));
  JanB : Person =
         (FirstName : 'Jan';
          LastName : 'Bielecki';
          Hair : Black;
          Age : 44);
  Family : array[1..3] of Person =
           ((FirstName : 'Jan';
             LastName : 'Bielecki';
             Hair : Black;
             Age : 44),
            (FirstName : 'Ewa';
             LastName : 'Bielecka';
             Hair : Black;
             Age : 38),
            (FirstName : 'Iza';
             LastName : 'Bielecka';
             Hair : Black;
             Age : 3));
```

- Elementowi *Vector*[2] przypisano daną początkową *male*.
- Elementowi *Matrix*[*Fair*,3] przypisano daną początkową 5.
- Elementowi *ThreeD*[*false*,*true*,*true*] przypisano daną początkową *Fair*.
- Komponentowi *JanB.Age* przypisano daną początkową 44.
- Komponentowi *Family*[2].*Age* przypisano daną początkową 38.
- Gdyby deklaracji zmiennej *Vector* nadano postać
  Vector : **array**[1..3] **of** *Sex* =
  (*male*,*female*);
to byłaby ona błędna, ponieważ zmienna ta jest tablicą 3-elementową, natomiast lista inicjatorów zawarta w nawiasach okrągłych składa się tylko z 2 elementów.
- Gdyby deklaracji zmiennej *JanB* nadano postać
  JanB : *Person* = (*FirstName* : *'Janek'*);
to byłaby ona poprawna i równoważna deklaracji
  JanB : *Person* = (*FirstName* : *'Jan'*);
- Gdyby deklaracji zmiennej *JanB* nadano postać
  JanB : *Person* = (*LastName* : *'Bielecki'*);
to byłaby ona niepoprawna. ■

Przypisania danych początkowych polom rekordu z wariantami są sensowne tylko wówczas, gdy dotyczą tych pól wariantu, które do niego należą. Wymaganie to nie jest kontrolowane przez kompilator.

Przykład

```
type
  Sex = (male,female);
  Color = (Black,Red,Blond);
  Person = record
             Name : string[20];
             case Gender : Sex of
               male : (Height : byte);
               female : (Hair : Color)

           end;
const
  Trio : array[1..3] of Person =
          ((Name : 'Ewa'),
           (Name : 'Jan'; Gender : male),
           (Name : 'Iza'; Gender : female;
                          Hair : Black));
```

- Komponentowi *Trio*[2].*Gender* tablicy rekordów *Trio* przypisano daną początkową *male*.
- Inicjator o postaci
  (*Name* : *'Iza'*; *Gender* : *male*; *Hair* : *Black*)

chociaż mało sensowny, jest uznawany za poprawny. ■

15. FUNKCJE I PROCEDURY

Funkcje i *procedury* są obiektami opisującymi dobrze zdefiniowane fragmenty algorytmu realizowanego przez program. Z tego względu będą nazywane *podprogramami*.

W odróżnieniu od innych obiektów strukturalnych, jak np. instrukcji wyboru, wykonanie podprogramu wymaga jego wywołania, tj. stosownie do sytuacji, posłużenia się instrukcją wywołania procedury albo wywołaniem funkcji. *Instrukcja wywołania procedury* może wystąpić w każdym miejscu programu, w którym może być użyta np. instrukcja pusta, natomiast *wywołanie funkcji* może być użyte jedynie w wyrażeniu, gdzie występuje pod postacią tzw. *nazewnika funkcji*.

Podprogramy, podobnie jak zmienne, wymagają deklarowania. Deklaracje podprogramów mogą być umieszczane w części deklaracyjnej bloku. *Deklaracja podprogramu*, nazywana niekiedy jego *definicją*, składa się z *nagłówka* oraz z *bloku* stanowiącego *ciało podprogramu*. *Nagłówek procedury* składa się ze słowa kluczowego **procedure**, po którym następuje nazwa procedury, ujęty w nawiasy okrągłe wykaz parametrów procedury oraz średnik. *Nagłówek funkcji* składa się ze słowa kluczowego **function**, po którym następuje nazwa funkcji, ujęty w nawiasy okrągłe wykaz parametrów funkcji, dwukropek, określenie typu rezultatu funkcji oraz średnik. Jeśli wykaz parametrów podprogramu jest pusty, to jest opuszczany wraz z otaczającymi go nawiasami.

Wymaga się, aby podczas wykonywania funkcji została wykonana instrukcja przypisania, w której z lewej strony symbolu przypisania występuje nazwa funkcji, a z prawej wyrażenie zgodne z typem rezultatu funkcji. Dana reprezentowana przez wyrażenie, które wystąpiło w ostatnim tak wykonanym przypisaniu, stanowi rezultat funkcji. Wykonanie funkcji nie musi ograniczyć się do udostępnienia rezultatu. Może ono także

spowodować zmianę pewnych argumentów jej wywołania.

Składnia

```
deklaracja-podprogramu:
  deklaracja-procedury
  deklaracja-funkcji
deklaracja-procedury:
  procedure nazwa-procedury ( wykaz-parametrów ) ;
  procedure nazwa-procedury
deklaracja-funkcji:
  function nazwa-funkcji ( wykaz-parametrów )
                         : typ-rezultatu ;
  function nazwa-funkcji : typ-rezultatu ;
nazwa-procedury:
  identyfikator
nazwa-funkcji:
  identyfikator
typ-rezultatu:
  identyfikator-typu
identyfikator-typu:
  identyfikator
```

Elementy wykazu parametrów podprogramu są oddzielone średnikami. Każdy element wykazu zawiera listę identyfikatorów parametrów, po której następuje dwukropek i identyfikator typu parametrów danej listy.

W chwili wywołania podprogramu następuje skojarzenie parametrów podprogramu z argumentami jego wywołania. Liczba argumentów wywołania musi być równa liczbie parametrów, a skojarzenia parametrów z argumentami dokonują się w kolejności ich wystąpienia w nagłówku podprogramu i w wywołaniu.

Skojarzenie parametru z argumentem może dokonać się przez wartość albo przez wskazanie. W pierwszym przypadku parametr jest traktowany jak lokalna zmienna podprogramu, której w chwili rozpoczęcia jego wykonywania (dla danego wywołania) przypisano daną reprezentowaną przez argument. W drugim przypadku parametr jest traktowany tak, jakby reprezentował argument. Ma to ten skutek, że każda operacja dotycząca parametru jest realizowana tak, jakby dotyczyła argumentu. Posłużenie się tym rodzajem skojarzenia parametru z argumentem wymaga poprzedzenia listy identyfikatorów parametrów słowem kluczowym var.

Szczególnym rodzajem skojarzenia przez wskazanie jest skojarzenie parametru z argumentem, który jest nazwą podprogramu. Jednym z ograniczeń języka Turbo Pascal w stosunku do standardu języka Pascal jest brak tego rodzaju skojarzenia.

Składnia

element-wykazu-parametrów:
  lista-nazw-parametrów : *oznaczenie-typu*
  var *lista-nazw-parametrów* : *oznaczenie-typu*
oznaczenie-typu:
  identyfikator-typu
identyfikator-typu:
  identyfikator
nazwa-parametru:
  identyfikator

Przykłady

a. Procedura z jednym parametrem, skojarzenie przez wartość

```
program FiveMessages;
var
  Tally : byte;
procedure SlowDown(Count : integer);
begin
   for Count := Count downto 1 do
end;
begin
   for Tally := 1 to 5 do begin
     Writeln('Wake up');
     SlowDown(400)
   end
end.
```

- Program *FiveMessages* zawiera 2 deklaracje: deklarację zmiennej *Tally* i deklarację procedury *SlowDown*.
- Wykazem, a zarazem elementem wykazu parametrów, jest napis

  Count : *integer*

- Skojarzenie parametru *Count* z argumentem 400 dokona się przez wartość.
- Operacje na parametrze *Count* są w istocie operacjami na pewnej lokalnej zmiennej procedury *SlowDown*. Początkową wartością tej zmiennej jest 400.

b. Procedura z dwoma parametrami, skojarzenie przez wskazanie

```
program ConvertAndSwap;
var
  theFloat : real;
  theFixed : integer;
procedure Swap(var Float : real;
               var Fixed : integer);
```

```
var
  Temp : real;
begin
  Temp := Float;
  Float := Fixed;
  Fixed := trunc(Temp + 0.5)
end;
begin
  theFloat := 12.8;
  theFixed := 10;
  Writeln(theFloat,theFixed :3);
  Swap(theFloat,theFixed :3);
  Writeln(theFloat,theFixed :3)
end.
```

● Program *ConvertAndSwap* zawiera 6 deklaracji: deklaracje zmiennych *theFloat* i *theFixed*, deklarację procedury *Swap*, deklaracje parametrów *Float* i *Fixed* oraz deklarację zmiennej *Temp*.

● Wykaz parametrów procedury *Swap* składa się z 2 elementów oddzielonych średnikiem.

● Skojarzenia parametrów procedury z argumentami wywołania dokonują się przez wskazanie.

● Operacje na parametrze *Float* są w istocie operacjami na argumencie *theFloat*, a operacje na parametrze *Fixed* są operacjami na argumencie *theFixed*.

● Wykonanie programu powoduje wyprowadzenie 2 wierszy tekstu. W pierwszym znajdą się liczby 12.8 i 10, a w drugim liczby 10 i 13.

● Gdyby nagłówkowi procedury *Swap* nadano postać

```
procedure Swap(Float : real;
               var Fixed : integer);
```

niczego poza tym nie zmieniając, to zostałyby wyprowadzone liczby 12.8 i 10 oraz 12.8 i 13. Wynika to z faktu, że po wprowadzeniu takiej zmiany skojarzenie parametru *Float* z argumentem *theFloat* dokonałoby się przez wartość.

c. Najkrótszy program zawierający deklarację i wywołanie procedury bezparametrowej

```
procedure P;
begin
end;
begin
  P
end.
```

d. Funkcja z trzema parametrami

```
program MultiplyDivide;
var
  Product : integer;
function Divide(SourceOne,SourceTwo : byte;
                var Target : integer) : real;
begin
  Target := SourceOne * SourceTwo;
  Divide := SourceOne / SourceTwo
end;
begin
  Writeln(Divide(12,4,Product),Product :3)
end.
```

● Parametry *SourceOne* i *SourceTwo* są skojarzone z argumentami 12 i 4 przez wartość. Parametr *Target* jest skojarzony z argumentem *Product* przez wskazanie. Rezultatem funkcji jest dana typu *real* przypisana zmiennej o nazwie *Divide*.

● Wykonanie programu powoduje wyprowadzenie 1 wiersza tekstu zawierającego liczby 3.0 i 48.

● Gdyby nagłówkowi funkcji *Divide* nadano postać

```
function Divide(SourceOne,SourceTwo : byte;
                Target : integer) : real;
```

to program byłby niepoprawny z powodu nie przypisania zmiennej *Product* żadnej danej.

● Gdyby nagłówkowi funkcji *Divide* nadano natomiast postać

```
function Divide(var SourceOne,SourceTwo : byte;
                var Target : integer) : real;
```

to program byłby niepoprawny, ponieważ z parametrami zadeklarowanymi do skojarzenia przez nazwę próbowano by skojarzyć argumenty nie będące nazwami zmiennych, lecz wyrażeniami, tu 12 i 4.

e. Najkrótszy program zawierający deklarację i wywołanie funkcji bezparametrowej

```
function F : byte;
begin
  F := 0
end;
begin
  Write(F)
end.
```

f. Program zawierający parametry podprogramu reprezentujące podprogramy

```
program VeryComplicated;
procedure Ext(procedure Par;Flag : boolean);
var
  Fix : byte;
procedure Int;
begin
  Fix := Fix + 1;
end;
begin {Ext}
  Fix := 0;
  if Flag then Ext(Int,false)
          else Par;
  Writeln(Fix)
end;
begin {VeryComplicated}
  Out(Out,true)
end.
```

● Przytoczony program zawiera te konstrukcje języka standardowego, które nie zostały implementowane w języku Turbo Pascal. Z tego powodu jest to program błędny. ■

Kontrolowane przez kompilator wymaganie zgodności typu parametru z typem argumentu nie dotyczy tzw. *parametrów bez typu*, zadeklarowanych bez oznaczenia typu. Przyjmuje się bowiem, że nie ujawniony typ takiego parametru jest zgodny z typem dowolnego, skojarzonego z nim argumentu. To rozszerzenie języka, często w połączeniu z użyciem opisanego w rozdz. 19 słowa kluczowego **absolute** umożliwia w wielu przypadkach opracowywanie bardziej efektywnych programów.

Przykład

```
program MoveArray;
var
  SourceArray : array[1..3,1..4] of integer;
  TargetArray : array[1..2,1..6] of integer;
...
procedure MoveBytes(var Source,Target;
                    Count : integer);
var
  Index : integer;
type
  ByteField = array[1..MaxInt] of byte;
var
  Src : ByteField absolute Source;
```

```
  Trg : ByteField absolute Target;
begin
  for Index := 1 to Count do
    Trg[Index] := Src[Index]
end;
...
begin
  ...
  MoveBytes(SourceArray,TargetArray,24);
  ...
end.
```

● Tablice *SourceArray* i *TargetArray* są różnych typów, zatem wykonanie instrukcji

```
TargetArray := SourceArray
```

jest zabronione. Rezultat ten można jednak uzyskać posługując się procedurą *MoveBytes*.

● W procedurze tej *Source* i *Target* są parametrami bez typu. Lokalne zmienne *Src* i *Trg* typu *ByteField* zostają odwzorowane na obszary pamięci zajmowane przez tablice skojarzone z tymi parametrami. ■

Program, w którym parametr jawnie określonego typu jest skojarzony z argumentem innego typu, jest błędny. Wymaganie zgodności typu parametru i argumentu może zostać osłabione w odniesieniu do parametrów i argumentów typu łańcuchowego. Przyjęto bowiem, że w zasięgu działania dyrektywy kompilatora {$V-} każdy parametr typu łańcuchowego jest zgodny ze skojarzonym z nim argumentem typu łańcuchowego. Umożliwia to konstruowanie procedur do przetwarzania danych łańcuchowych dowolnych typów.

Przykład

```
program CountSpaces;
type
  OneLine = string[80];
function SpaceCount(SourceText : OneLine) : integer;
var
  Count,Index : integer;
begin
  Count := 0;
  for Index := 1 to Length(SourceText) do
    if SourceText[Index] = ' ' then
      Count := Count + 1;
  SpaceCount := Count
end;
```

```
begin
  ...
  Writeln(SpaceCount(
       {$V-}  'To be or not to be'  {$V+}
                        ));
  ...
end.
```

● Parametr *SourceText* jest typu **string**[80], natomiast skojarzony z nim argument jest typu **string**[18].

● Wywołanie funkcji *SpaceCount* jest poprawne, ponieważ skojarzenie parametru z argumentem odbywa się w zasięgu dyrektywy kompilatora {$V-}.

● Wykonanie instrukcji wywołania procedury *Writeln* powoduje wyprowadzenie liczby 5. ■

Podobnie jak w innych językach algolopodobnych, podprogramy zapisane w języku Turbo Pascal mogą zostać przygotowane w wersji rekurencyjnej. Niejednokrotnie umożliwia to uproszczenie algorytmu.

Przykład

```
function Fibonacci(Index : byte) : integer;
begin
  if Index < 3 then
     Fibonacci := 1
  else
     Fibonacci := Fibonacci(Index - 1) +
                  Fibonacci(Index - 2)
end;
```

● Przytoczona funkcja służy do wyznaczania wartości wyrazów ciągu Fibonacciego. W ciągu tym pierwsze dwa wyrazy mają wartość 1, natomiast każdy z następnych jest sumą dwóch bezpośrednio go poprzedzających. ■

W tych przypadkach, gdy posłużenie się rekurencją wymaga zadeklarowania np. dwóch procedur, z których pierwsza wywołuje drugą, a druga wywołuje pierwszą, niemożliwe jest takie uporządkowanie deklaracji, aby każde odwołanie dotyczyło obiektu już zadeklarowanego. W takiej sytuacji konieczne jest posłużenie się *deklaracją zapowiadającą*, składającą się z nagłówka podprogramu, po którym następuje średnik i słowo kluczowe **forward**.

W dogodnym miejscu po deklaracji zapowiadającej powinna wystąpić właściwa deklaracja podprogramu. Jej nagłówek nie zawiera już wykazu parametrów ujętego w nawiasy, a dla funkcji – także dwukropka i następującego po nim oznaczenia typu.

Przykład

```
function FunOne(var Source,Target : byte) : char;
  forward;
function FunTwo(Count : byte) : real;
begin
  ...
  Write(FunOne(Count,2*Count));
  ...
end;
function FunOne;
begin
  ...
  Write(FunTwo(Source + Target));
  ...
end;
```

● Ponieważ w ciele definicji funkcji *FunTwo* wystąpiło odwołanie do funkcji *FunOne*, a w ciele funkcji *FunOne* wystąpiło odwołanie do funkcji *FunTwo*, konieczne było posłużenie się deklaracją zapowiadającą, w której ciało funkcji *FunOne* zostało zastąpione napisem **forward**.

● Oczywiście nic nie stałoby na przeszkodzie, aby deklaracje przytoczonych funkcji wystąpiły w odwrotnej kolejności. W takim przypadku deklaracja zapowiadająca dotyczyłaby funkcji *FunTwo* i miałaby postać

```
function FunTwo(Count : byte) : real;
```

■

Określenie *podprogram standardowy* dotyczy tych podprogramów, które są dostępne bez jawnego definiowania. Wiele takich podprogramów opisano już uprzednio. Należą do nich podprogramy służące do wykonywania operacji na danych łańcuchowych, podprogramy do dynamicznego zarządzania pamięcią operacyjną oraz podprogramy realizujące operacje wejścia/wyjścia.

Uzupełnieniem tego obszernego zbioru podprogramów są procedury ekranowe i specjalne, funkcje arytmetyczne, skalarne i do wykonywania konwersji, jak również dość liczna grupa funkcji pomocniczych.

16.1. PROCEDURY EKRANOWE

Procedury ekranowe są dostępne w tych implementacjach, w których dokonano ich instalacji.

Procedura ClrEol

Wywołanie: *ClrEol*

Procedura *ClrEol* jest bezparametrowa.

Wykonanie procedury *ClrEol* powoduje zlokalizowanie wiersza, w którym znajduje się kursor, a następnie zastąpienie spacjami wszystkich znaków tego wiersza, począwszy od znaku wyróżnionego przez kursor. Pozycja kursora nie ulega zmianie.

Procedura ClrScr

Wywołanie: *ClrScr*

Procedura *ClrScr* jest bezparametrowa.

Wykonanie procedury *ClrScr* powoduje tzw. "wyzerowanie" ekranu, tj. zapełnienie go znakami spacji. Kursor zostanie umieszczony w lewym górnym rogu ekranu. W zależności od sposobu zainstalowania procedury czynności tej może towarzyszyć zmiana parametrów ustalonych za pomocą opisanych dalej procedur *LowVideo* i *NormVideo*.

Procedura CrtInit

Wywołanie: *CrtInit*

Procedura *CrtInit* jest bezparametrowa.

Wykonanie procedury *CrtInit* powoduje skierowanie do monitora łańcucha znaków inicjujących, określonego podczas instalowania systemu Turbo Pascal.

Procedura CrtExit

Wywołanie: *CrtExit*

Procedura *CrtExit* jest bezparametrowa.

Wykonanie procedury *CrtExit* powoduje skierowanie do monitora łańcucha znaków kończących, określonego podczas instalowania systemu Turbo Pascal.

Procedura DelLine

Wywołanie: *DelLine*

Procedura *DelLine* jest bezparametrowa.

Wykonanie procedury *DelLine* powoduje zlokalizowanie wiersza, w którym znajduje się kursor, a następnie usunięcie go i przemieszczenie wierszy znajdujących się poniżej o jeden wiersz do góry. Po wykonaniu tej czynności najniższy wiersz ekranu staje się pusty. Pozycja kursora nie ulega zmianie.

Procedura InsLine

Wywołanie: *InsLine*

Procedura *InsLine* jest bezparametrowa.

Wykonanie procedury *InsLine* powoduje zlokalizowanie wiersza, w którym znajduje się kursor, a następnie przemieszczenie tego wiersza, wraz z wierszami za nim następującymi, o jeden wiersz do dołu. Powoduje to umieszczenie na ekranie pustego wiersza i usunięcie z ekranu najniżej położonego z przemieszczanych wierszy. Pozycja kursora nie ulega zmianie.

Procedura GotoXY

Wywołanie: *GotoXY(xPos,yPos)*

Przyjmuje się, że *xPos* i *yPos* są wyrażeniami typu *integer*.

Wykonanie procedury *GotoXY* powoduje ustawienie kursora w takiej pozycji, że wyróżnia on znak o współrzędnych (*xPos*, *yPos*). Zakłada się, że lewy górny narożnik ekranu ma współrzędne (1,1); odcięte są liczone od lewej do prawej, a rzędne od góry do dołu. Jeśli wartość wyrażenia *xPos* albo *yPos* wykracza poza rozmiar wiersza albo kolumny, to każda z nich jest traktowana modulo rozmiar.

Procedura LowVideo

Wywołanie: *LowVideo*

Procedura *LowVideo* jest bezparametrowa.

Wykonanie procedury *LowVideo* powoduje, że aż do najbliższego wykonania procedury *NormVideo* znaki wyprowadzane na ekran są poddane inwersji.

Procedura NormVideo

Wywołanie: *NormVideo*

Procedura *NormVideo* jest bezparametrowa.

Wykonanie procedury *NormVideo* powoduje przywrócenie normalnego (por. procedura *LowVideo*) sposobu wyświetlania znaków.

16.2. PROCEDURY SPECJALNE

Procedura Delay

Wywołanie: *Delay(Time)*

Przyjmuje się, że *Time* jest wyrażeniem typu *integer*.

Wykonanie procedury *Delay* powoduje wstrzymanie wykonywania programu na okres (w przybliżeniu) *Time* milisekund. Jeśli wyrażenie *Time* ma wartość ujemną, to traktuje się je tak, jakby miało wartość 0.

Procedura Fill

Wywołanie: *Fill(Var,Cnt,Val)*

Przyjmuje się, że *Var* jest nazwą zmiennej dowolnego typu, *Cnt* jest wyrażeniem typu *integer*, a *Val* jest wyrażeniem typu *char* albo *byte*.

Wykonanie procedury *Fill* powoduje umieszczenie w obszarze, którego początek przydzielono zmiennej *Var*, *Cnt* danych bajtowych mających taką samą reprezentację jak dana bajtowa reprezentowana przez *Val*.

Procedura Exit

Wywołanie: *Exit*

Procedura *Exit* jest bezparametrowa.

Wykonanie procedury *Exit* ma taki sam skutek, jak wykonanie instrukcji przejścia do niejawnej instrukcji pustej poprzedzającej słowo kluczowe **end** tego bloku, w którym wywołanie to nastąpiło. Powoduje to zatem zakończenie wykonywania bloku, a jeśli jest nim blok programu, także zakończenie wykonywania całego programu.

Procedura Halt

Wywołanie: *Halt*

Procedura *Halt* jest bezparametrowa.

Wykonanie procedury *Halt* powoduje zakończenie wykonywania programu.

Procedura Move

Wywołanie: *Move(Src,Trg,Cnt)*

Przyjmuje się, że *Src* i *Trg* są nazwami zmiennych dowolnego typu, a *Cnt* jest wyrażeniem typu *integer*.

Wykonanie procedury *Move* powoduje przepisanie z obszaru, którego początek przydzielono zmiennej *Src*, do obszaru, którego początek przydzielono zmiennej *Trg*, *Cnt* bajtów pamięci. Przepisanie jest poprawne nawet wtedy, gdy wspomniane obszary częściowo albo w całości pokrywają się.

Procedura Randomize

Wywołanie: *Randomize*

Procedura *Randomize* jest bezparametrowa.

Wykonanie procedury *Randomize* powoduje zainicjowanie generatora liczb przypadkowych daną o wartości przypadkowej.

16.3. FUNKCJE ARYTMETYCZNE

Funkcja Abs

Wywołanie: *Abs(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *real* albo *integer*.

Rezultatem funkcji *Abs* jest dana takiego samego typu jak typ argumentu, o wartości równej wartości bezwzględnej argumentu, stanowiąca wartość bezwzględną danej reprezentowanej przez *Num*. Na przykład

Abs(-2) = 2
Abs(Pi) = 3.1415926536

Funkcja ArcTan

Wywołanie: *ArcTan(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *real* albo *integer*.

Rezultatem funkcji *ArcTan* jest dana typu *real* stanowiąca arcus tangens danej reprezentowanej przez *Num*. Przykładowo

ArcTan(0) = 0.0
ArcTan(1.0) = *Pi/4*

Funkcja Cos

Wywołanie: *Cos(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *real* albo *integer*.

Rezultatem funkcji *Cos* jest dana typu *real* stanowiąca cosinus danej reprezentowanej przez *Num*. Na przykład

Cos(0) = 1.0
Cos(Pi) = -1.0

Funkcja Exp

Wywołanie: *Exp(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *real* albo *integer*.

Rezultatem funkcji *Exp* jest dana typu *real* uzyskana z podniesienia do potęgi *Num* podstawy logarytmów naturalnych

e = 2.7182818285. Na przykład
Exp(0) = 1.0
Exp(-1.0) = 0.36787944117

Funkcja Frac

Wywołanie: *Frac*(*Num*)

Przyjmuje się, że *Num* jest wyrażeniem typu *real* albo *integer*.

Rezultatem funkcji *Frac* jest dana typu *real* stanowiąca część ułamkową danej reprezentowanej przez *Num*. Na przykład
Frac(3) = 0.0
Frac(*Pi*) = *Pi* - 3.0

Funkcja Int

Wywołanie: *Int*(*Num*)

Przyjmuje się, że *Num* jest wyrażeniem typu *real* albo *integer*.

Rezultatem funkcji *Int* jest dana typu *real* stanowiąca część całkowitą danej reprezentowanej przez *Num*. Na przykład
Int(2) = 2.0
Int(-*Pi*) = -3.0

Funkcja Ln

Wywołanie: *Ln*(*Num*)

Przyjmuje się, że *Num* jest wyrażeniem typu *real* albo *integer*.

Rezultatem funkcji *Ln* jest dana typu *real* stanowiąca logarytm naturalny danej reprezentowanej przez *Num*. Przykładowo
Ln(1) = 0.0
Ln(3.0) = 1.0986122887

Funkcja Sin

Wywołanie: *Sin*(*Num*)

Przyjmuje się, że *Num* jest wyrażeniem typu *real* albo *integer*.

Rezultatem funkcji *Sin* jest dana typu *real* stanowiąca sinus danej reprezentowanej przez *Num*. Na przykład
Sin(0) = 0.0
Sin(*Pi*/2) = 1.0

Funkcja Sqr

Wywołanie: *Sqr(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *real* albo *integer*.

Rezultatem funkcji *Sqr* jest dana takiego samego typu jak typ argumentu, stanowiąca kwadrat danej reprezentowanej przez *Num*. Na przykład

Sqr(2) = 4
Sqr(2.0) = 4.0

Funkcja Sqrt

Wywołanie: *Sqrt(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *real* albo *integer*.

Rezultatem funkcji *Sqrt* jest dana typu *real*, stanowiąca pierwiastek kwadratowy danej reprezentowanej przez *Num*. Przykładowo

Sqrt(4) = 2.0
Sqrt(4.0) = 2.0

16.4. FUNKCJE SKALARNE

Funkcja Odd

Wywołanie: *Odd(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *integer*.

Rezultatem funkcji *Odd* jest dana typu *boolean*, wyrażająca wartość logiczną zdania "wyrażenie *Num* reprezentuje daną o wartości nieparzystej". Na przykład

Odd(3) = *true*
Odd(4) = *false*

Funkcja Pred

Wywołanie: *Pred(Num)*

Przyjmuje się, że *Num* jest wyrażeniem dowolnego typu porządkowego.

Rezultatem funkcji *Pred* jest dana takiego samego typu jak typ argumentu, stanowiąca poprzednik danej reprezentowanej przez *Num* w jego typie porządkowym. Na przykład

Pred('B') = *'A'*
Pred(-20) = -21

Funkcja Succ

Wywołanie: *Succ(Num)*

Przyjmuje się, że *Num* jest wyrażeniem dowolnego typu porządkowego.

Rezultatem funkcji *Succ* jest dana takiego samego typu jak typ argumentu, stanowiąca następnik danej reprezentowanej przez *Num* w jego typie porządkowym. Na przykład

Succ(false) = *true*
Succ(0) = 1

16.5. FUNKCJE DO WYKONYWANIA KONWERSJI

Funkcja Chr

Wywołanie: *Chr(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *integer*.

Rezultatem funkcji *Chr* jest dana typu *char*, reprezentująca znak o kodzie *Num*. Na przykład

Chr(65) = *'A'*
Chr(27) = ^[

Funkcja Ord

Wywołanie: *Ord(Num)*

Przyjmuje się, że *Num* jest wyrażeniem dowolnego typu porządkowego.

Rezultatem funkcji *Ord* jest dana typu *integer*, określająca numer porządkowy danej reprezentowanej przez *Num*, w jego typie porządkowym. Na przykład

Ord(true) = 1
Ord(-2) = -2

Funkcja Round

Wywołanie: *Round(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *real*.

Rezultatem funkcji *Round* jest dana typu *integer* o wartości najbliższej wartości danej reprezentowanej przez *Num*. Na przykład

Round(5.5) = 6
Round(-1.5) = -2

Funkcja Trunc

Wywołanie: *Trunc(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *real*.

Rezultatem funkcji *Trunc* jest dana typu *integer* stanowiąca część całkowitą danej reprezentowanej przez *Num*. Na przykład

Trunc(3.14) = 3
Trunc(-2.9) = -2

16.6. FUNKCJE POMOCNICZE

Funkcja Hi

Wywołanie: *Hi(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *integer*.

Rezultatem funkcji *Hi* jest dana typu *integer* o wartości bardziej znaczącego bajtu danej reprezentowanej przez *Num*. Na przykład

Hi(256) = 1
Hi(-1) = 255

Funkcja KeyPressed

Wywołanie: *KeyPressed*

Funkcja *KeyPressed* jest bezparametrowa.

Rezultatem funkcji *KeyPressed* jest dana typu *boolean* wyrażająca wartość logiczną zdania "w buforze wejściowym konsoli znajduje się nie wprowadzony jeszcze znak".

Funkcja Lo

Wywołanie: *Lo(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *integer*.

Rezultatem funkcji *Lo* jest dana typu *integer* o wartości mniej znaczącego bajtu danej reprezentowanej przez *Num*. Na przykład

Lo(256) = 0
Lo(-1) = 255

Funkcja Random

Wywołanie: *Random*
Random(Num)

Przyjmuje się, że *Num* jest wyrażeniem typu *integer*.

Rezultatem funkcji *Random* w wersji bezparametrowej jest dana typu *real* o wartości przypadkowej większej lub równej 0.0 i mniejszej niż 1.0 Rezultatem funkcji *Random* w wersji z parametrem jest dana typu *integer* o wartości przypadkowej większej lub równej 0 i mniejszej niż *Num*.

Funkcja ParamCount

Wywołanie: *ParamCount*

Funkcja *ParamCount* jest bezparametrowa.

Rezultatem funkcji *ParamCount* jest dana typu *integer* określająca liczbę parametrów przekazanych programowi w chwili jego wywołania. Zakłada się, że parametry zostały oddzielone spacjami lub znakami tabulacji.

Funkcja ParamStr

Wywołanie: *ParamStr(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *integer*.

Rezultatem funkcji *ParamStr* jest dana łańcuchowa reprezentująca ten parametr przekazany programowi w chwili jego wywołania, który ma numer *Num*. Zakłada się, że pierwszy parametr ma numer 1.

Funkcja SizeOf

Wywołanie: *SizeOf(Nam)*

Przyjmuje się, że *Nam* jest nazwą zmiennej albo nazwą typu.

Rezultatem funkcji *SizeOf* jest dana typu *integer* określająca liczbę bajtów pamięci przydzielonych zmiennej *Nam* albo liczbę bajtów pamięci niezbędnych do reprezentowania danej typu *Nam*. Na przykład

SizeOf(boolean) = 1
SizeOf(integer) = 2

Funkcja Swap

Wywołanie: *Swap(Num)*

Przyjmuje się, że *Num* jest wyrażeniem typu *integer*.

Rezultatem funkcji *Swap* jest dana typu *integer*, której bardziej znaczący bajt ma wartość *Lo(Num)*, a mniej znaczący bajt ma wartość *Hi(Num)*. Na przykład

Swap(256) = 1
Swap(1) = 256

Funkcja UpCase

Wywołanie: *UpCase(Chr)*

Przyjmuje się, że *Chr* jest wyrażeniem typu *char*.

Rezultatem funkcji *UpCase* jest dana typu *char*, która jeśli *Chr* reprezentuje literę - jest literą dużą, a jeśli nie reprezentuje litery - jest taka sama jak dana reprezentowana przez *Chr*. Na przykład

UpCase('a') = 'A'
UpCase('.') = '.'

Przykład

```
program PrintArguments;
var
  Count : byte;
begin
  Write('Arguments are:');
  for Count := 1 to ParamCount do
    Write(' ',ParamStr(Count))
end.
```

- Jeśli program *ParamArguments* zostanie wywołany z argumentami

  Jan Ewa Iza

to jego wykonanie spowoduje wyprowadzenie napisu

  Arguments are: Jan Ewa Iza

- Jeśli zostanie wywołany bez argumentów, to jego wykonanie spowoduje wyprowadzenie napisu

  Arguments are: ■

Jedną z ważniejszych dyrektyw kompilatora jest dyrektywa włączenia zbioru. Ma ona postać

{$I *name* }

gdzie *name* jest nazwą zbioru.

Zinterpretowanie takiej dyrektywy powoduje zastąpienie jej zawartością zbioru o nazwie *name*. Umożliwia to włączenie do programu uprzednio przygotowanych zestawów deklaracji albo bibliotek podprogramów w postaci źródłowej. Wymaga się jedynie, aby zbiór włączany nie zawierał dyrektyw włączania zbiorów.

Należy uwzględnić także to, że jeśli nazwa *name* nie zawiera trzyliterowego przyrostka, to nawias klamrowy kończący dyrektywę włączenia zbioru powinien być oddzielony od *name* przynajmniej jedną spacją. W przeciwnym razie klamra zamykająca zostanie uznana za część nazwy zbioru.

Przykład

Jeśli przyjąć, że w zbiorze *Silnia.doc* znajduje się tekst

```
function Silnia(Num : byte) : integer
begin
  if Num < 2 then Silnia := 1
             else Silnia := Num * Silnia(Num - 1)
end;
```

to skompilowanie programu

```
program Demo;
{$I Silnia.doc }
begin
  Writeln(Silnia(5))
end.
```

będzie miało taki sam skutek jak skompilowanie programu

```
program Demo;
function Silnia(Num : byte) : integer;
begin
  if Num < 2 then Silnia := 1
             else Silnia := Num * Silnia(Num - 1)
end;
begin
  Writeln(Silnia(5))
end.
```

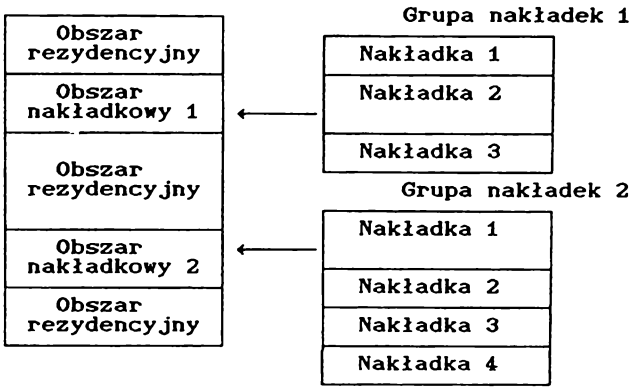
Istotnym ograniczeniem systemu Turbo Pascal jest założenie, że rozmiar kodu programu wynikowego oraz rozmiar jego danych nie przekracza 64K bajtów pamięci operacyjnej. Osłabienie skutków tego ograniczenia można uzyskać posługując się *nakładkowaniem*.

Nakładkowaniu mogą podlegać zarówno funkcje, jak i procedury. Wymaga się jedynie, aby podprogramy nakładkowane nie były wywoływane rekurencyjnie oraz aby nie były przedmiotem deklaracji zapowiadających. Nie stanowi to istotnego utrudnienia, ponieważ nic nie stoi na przeszkodzie zadeklarowania podprogramu nienakładkowego wywołującego podprogram, który miał być np. wywołany rekurencyjnie.

Pewne trudności mogą powstać podczas wyszukiwania błędów w podprogramach nakładkowanych. Żeby nie wdawać się w opisy sposobów wykrywania takich błędów, wystarczy poprzestać na radzie, aby podprogramy zwykłe przekształcano w nakładkowane dopiero po ich uruchomieniu.

Zasadę nakładkowania przedstawiono na rys. 18.1. Polega ona na tym, że obszar pamięci operacyjnej przydzielony programowi wynikowemu zostaje podzielony na dwa rodzaje obszarów: obszary, w których zostanie ulokowana część rezydencyjna programu, i obszary, do których będą sprowadzane nakładki.

Każda *nakładka* składa się z grupy podprogramów, której przypisano *obszar nakładkowy*. W każdej chwili wykonywania programu w obszarze nakładkowym może znajdować się co najwyżej jeden podprogram grupy. Rozmiar obszaru nakładkowego jest tak dobrany, że mieści się w nim każdy z podprogramów grupy. Oznacza to, że oszczędność pamięci operacyjnej wynikająca z posłużenia się grupą nakładkową wyraża się różnicą między rozmiarem grupy a rozmiarem największej nakładki w grupie.



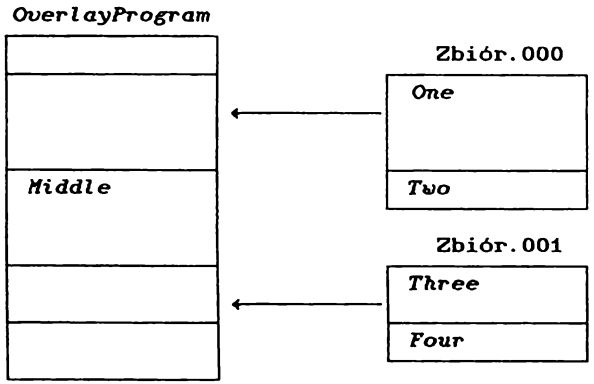
Rys.18.1. Zasada nakładkowania

Ceną, jaką płaci się za oszczędność pamięci operacyjnej, jest zakaz wzajemnego odwoływania się podprogramów grupy oraz konieczność wykonywania transmisji dyskowych podczas odwoływania się do tych programów, które należą do grupy nakładkowej, ale w chwili wywołania nie znajdują się w pamięci operacyjnej.

Zasady tworzenia grup nakładkowych są bardzo proste. W celu utworzenia grupy wystarczy poprzedzić słowem kluczowym **overlay** jedną lub więcej następujących po sobie deklaracji podprogramów. Wszystkie takie podprogramy będą wówczas stanowić grupę, zapamiętaną jako zbiór o nazwie programu głównego (dyrektywa M menu systemu) z trzycyfrowym przyrostkiem. Pierwsza grupa uzyska nazwę z przyrostkiem .000, druga z przyrostkiem .001 itd. Jeśli między podprogramami nakładkowanymi wystąpią podprogramy nienakładkowane, to nakładkowane znajdą się w rozłącznych grupach. Zilustrowano to na rys. 18.2, na którym przedstawiono strukturę programu wynikowego powstałego z programu

```
program OverlayProgram;
overlay procedure One;
begin
  ...
end;
overlay procedure Two;
begin
  ...
end;
```

```
procedure Middle;
begin
  ...
end;
overlay procedure Three;
begin
  ...
end;
overlay procedure Four;
begin
  ...
end;
begin
  ...
end.
```

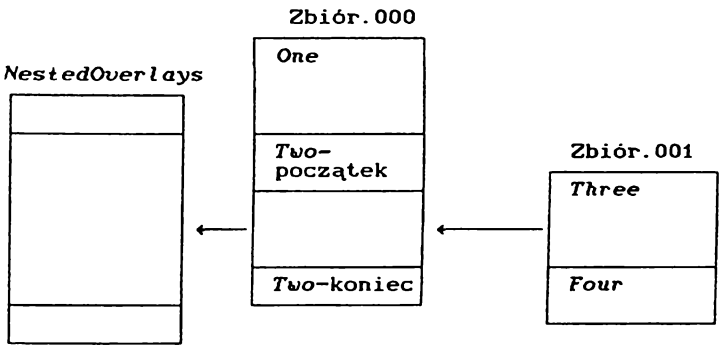


Rys.18.2. Struktura programu *OverlayProgram*

Zasady tworzenia grup nakładkowych nie wykluczają możliwości tworzenia nakładek w obrębie podprogramów nakładkowanych. Przedstawiono to na rys. 18.3 dla programu wynikowego powstałego z programu

```
program NestedOverlays;
overlay procedure One;
begin
  ...
end;
overlay function Two(Num : byte) : byte;
  overlay function Three : real;
```

```
    begin
      ...
    end;
    overlay procedure Four;
    begin
      ...
    end;
  begin {Two}
    ...
  end;
  begin {NestedOverlays}
    ...
  end.
```



Rys.18.3. Struktura programu *NestedOverlays*

System Turbo Pascal, rozumiany jako połączenie języka programowania i złączonego z nim edytora ekranowego, może być instalowany w środowisku różnych systemów operacyjnych. W każdym z nich udostępnia on dodatkowe możliwości, specyficzne dla obranego systemu operacyjnego i kompilatora. Do możliwości tych należą między innymi:

- Jawne przydzielanie miejsca zmiennym programu.
- Dodatkowe funkcje i procedury systemowe.
- Bezpośrednie wywoływanie podprogramów systemowych.
- Programowanie w języku asemblera.
- Aktywowanie innych programów.

Każda z nich zostanie tu przedstawiona w postaci, w jakiej obowiązuje w systemie CP/J.

19.1. JAWNE PRZYDZIELANIE MIEJSCA ZMIENNYM PROGRAMU

Użycie w deklaracji pewnej zmiennej słowa kluczowego **absolute** umożliwia zarezerwowanie miejsca dla tej zmiennej w miejscu przydzielonym innej zmiennej albo w ustalonym obszarze pamięci operacyjnej.

Deklaracja zawierająca słowo kluczowe **absolute** składa się z nazwy zmiennej deklarowanej, po której następuje dwukropek, określenie typu zmiennej, słowo kluczowe **absolute**, określenie miejsca, które ma zostać przydzielone zmiennej i średnik. Określenie miejsca może miec postać nazwy uprzednio zadeklarowanej albo postać adresu bezwzględnego wyrażonego za pomocą literału typu *integer* (Elwro 800 Jr) albo oddzielonych dwukropkiem numeru segmentu i przemieszczenia w segmencie (IBM PC).

Składnia

```
deklaracja-ze-słowem-absolute:
   nazwa-deklarowana : oznaczenie typu
                       absolute określenie-miejsca
nazwa-deklarowana:
      identyfikator
określenie-miejsca:
      nazwa-zmiennej
      literał
      nazwa-literału
nazwa-zmiennej:
      identyfikator
```

Przykład (dla Elwro 800 Jr)

```
var
  CharStr : string[6];
  StrLen : byte absolute CharStr;
  Buffer : array[0..15] of byte
           absolute $2000;
  function Fun(var IntVar : integer) : byte;
  var
    LSB : byte absolute IntVar;
  begin
    if LSB = Lo(IntVar) then ...
    ...
  end;
```

● Pierwszy bajt obszaru pamięci operacyjnej, przydzielonego zmiennej *CharStr*, pokrywa się z bajtem przydzielonym zmiennej *StrLen*. Odwołanie do *StrLen* reprezentuje zatem taką samą daną jak odwołanie *Lenght(CharStr)* oraz *ord(CharStr[0])*.

● Zmiennej *Buffer* zostanie przydzielony obszar pamięci operacyjnej rozpoczynający się od bajtu o adresie $2000.

● Zmiennej *LSB* zostanie przydzielony ten sam bajt pamięci operacyjnej, który został przydzielony mniej znaczącemu bajtowi zmiennej, z którą skojarzono parametr *IntVar*. Z tego powodu relacja zawarta w funkcji *Fun* będzie zawsze prawdziwa.

■

19.2. DODATKOWE FUNKCJE I PROCEDURY SYSTEMOWE

Funkcja Addr

Wywołanie: *Addr(Nam)*

Przyjmuje się, że *Nam* jest nazwą zmiennej albo podprogramu.

Rezultatem funkcji *Addr* jest dana wskazująca, określająca adres obszaru pamięci przydzielonego zmiennej albo podprogramowi *Nam*. Na przykład

Addr(Mem[2])

Procedura OvrDrive

Wywołanie: *OvrDrive(Drv)*

Przyjmuje się, że *Drv* jest wyrażeniem typu *integer*.

Wykonanie procedury *OvrDrive* powoduje, że w programie nakładkowanym nakładki będą poszukiwane w stacji dyskowej określonej przez *Drv* (0 = stacja domniemana,1 = A, 2 = B itd.). Na przykład

OvrDrive(2)

19.3. WYWOŁYWANIE PODPROGRAMOW SYSTEMOWYCH (ELWRO 800 JR)

Bezpośrednie wywoływanie podprogramów systemowych zapewniają procedury *Bdos* i *Bios* oraz funkcje *Bdos*, *BdosHL*, *Bios* i *BiosHL*.

Procedura Bdos

Wywołanie: *Bdos(Fun,Par)*
Bdos(Fun)

Przyjmuje się, że *Fun* oraz nie wymagane *Par* są wyrażeniami typu *integer*.

Wykonanie procedury *Bdos* powoduje załadowanie do rejestru *C* procesora mniej znaczącego bajtu danej reprezentowanej przez *Fun*, a następnie przejście do wykonywania programu począwszy od adresu 5 w celu zrealizowania funkcji systemu BDOS określonej przez *Fun*. Jeśli procedura jest wywoływana z dwoma argumentami, to przed wykonaniem wspomnianego skoku następuje umieszczenie danej reprezentowanej przez *Par* w rejestrze *DE*.

Funkcja Bdos

Wywołanie: *Bdos(Fun,Par)*
Bdos(Fun)

Przyjmuje sie, że *Fun* oraz nie wymagane *Par* są wyrażeniami typu *integer*.

Skutki wywołania funkcji *Bdos* są takie same jak skutki wywołania procedury *Bdos* z takimi samymi argumentami. Rezul-

tatem funkcji jest natomiast dana typu *byte* pozostawiana przez *Bdos* w rejestrze *A* procesora.

Funkcja BdosHL

Wywołanie: *BdosHL(Fun,Par)*
BdosHL(Fun)

Przyjmuje się, że *Fun* oraz nie wymagane *Par* są wyrażeniami typu *integer*.

Skutki wywołania funkcji *BdosHL* są takie same jak skutki wywołania funkcji *Bdos* z takimi samymi argumentami, z tą różnicą, że rezultatem funkcji jest dana typu *integer* pozostawiana przez BDOS w rejestrach *HL* procesora.

Procedura Bios

Wywołanie: *Bios(Fun,Par)*
Bios(Fun)

Przyjmuje się, że *Fun* oraz nie wymagane *Par* są wyrażeniami typu *integer*.

Wykonanie procedury *Bios* powoduje wykonanie funkcji podsystemu BIOS, która ma numer *Fun*. Jeśli posłużono się argumentem *Par*, to przed wykonaniem tej czynności w rejestrach *BC* procesora jest umieszczana dana reprezentowana przez *Par*.

Funkcja Bios

Wywołanie: *Bios(Fun,Par)*
Bios(Fun)

Przyjmuje się, że *Fun* oraz nie wymagane *Par* są wyrażeniami typu *integer*.

Skutki wywołania funkcji *Bios* są takie same jak skutki wywołania procedury *Bios* z takimi samymi argumentami. Rezultatem funkcji jest natomiast dana typu *byte* pozostawiana przez BIOS w rejestrze *A* procesora.

Funkcja BiosHL

Wywołanie: *BiosHL(Fun,Par)*

Przyjmuje się, że *Fun* oraz nie wymagane *Par* są wyrażeniami typu *integer*.

Skutki wywołania funkcji *BiosHL* są takie same jak skutki wywołania funkcji *Bios* z takimi samymi argumentami, z tą

różnicą, że rezultatem funkcji jest dana typu *integer* pozostawiana przez BIOS w rejestrach *HL* procesora.

19.4. PROGRAMOWANIE W JĘZYKU ASEMBLERA (ELWRO 800 JR)

Niekiedy zachodzi potrzeba dołączenia do programu napisanego w języku Turbo Pascal podprogramu napisanego w asemblerze albo przynajmniej posłużenia się wstawką asemblerową. Celowi temu służą konstrukcje językowe zawierające słowa kluczowe **external** i **inline**.

Jeśli zamiast bloku stanowiącego ciało podprogramu występuje konstrukcja

external *literał-typu-integer*

to oznacza to, że podprogram jest napisany w języku asemblerowym i znajduje się w pamięci operacyjnej pod adresem określonym przez literał. Wymaga się, aby wykonanie takiego podprogramu nie spowodowało zmiany rejestru *SP* (rozmiaru stosu).

Przykład

```
function Factorial(Arg : byte) : integer;
  external $6000;
```

● Część deklaracyjną i wykonawczą funkcji *Factorial* zastąpiono konstrukcją zawierającą słowo kluczowe **external**.

● Funkcja *Factorial* jest napisana w języku asemblerowym, a jej kod znajduje się w pamięci operacyjnej począwszy od adresu $6000. ■

O ile konstrukcja external jedynie wskazuje miejsce, w którym znajduje się podprogram asemblerowy, o tyle instrukcja inline umożliwia wygenerowanie kodu asemblerowego.

Instrukcja inline składa się ze słowa kluczowego **inline** bezpośrednio po którym następuje ujęty w nawiasy okrągłe ciąg elementów kodu. Elementy kodu są oddzielone kreskami ukośnymi, a każdy z nich składa się z elementów danych oddzielonych od siebie znakami + (plus) albo - (minus).

Elementem danych jest literał typu *integer*, identyfikator zmiennej, identyfikator podprogramu oraz wyrażone za pomocą znaku * (gwiazdka) oznaczenie licznika instrukcji.

Przykład

```
inline(20/$40/Fun-2/*+3)
```

■

Każdy element danych generuje jeden bajt albo jedno słowo (dwa bajty) kodu. Każda wygenerowana dana wynika z wykonania działań na elementach danych. Przyjmuje się, że iden-

tyfikator zmiennej albo podprogramu reprezentuje adres tej zmiennej albo podprogramu. Analogicznie przyjmuje się, że oznaczenie licznika instrukcji reprezentuje adres tego najbliższego bajtu pamięci, w którym zostanie umieszczony wygenerowany kod.

Jeśli element kodu składa się wyłącznie z literałów i separatorów, a jego wartość mieści się w zakresie 0..255, to zostanie wygenerowany jeden bajt kodu. Jeśli wartość wykracza poza ten zakres oraz jeśli element kodu zawiera nazwy zmiennych lub podprogramów albo odwołania do licznika instrukcji, to jest generowane słowo. W słowie tym bajt mniej znaczący jest generowany przed bajtem bardziej znaczącym.

Przytoczone domniemania liczby generowanych bajtów kodu mogą zostać zmienione, jeśli przed elementem kodu zostanie umieszczony znak < (mniejsze) albo > (większe). W pierwszym przypadku zostanie wygenerowany tylko mniej znaczący z tych bajtów, w których jest reprezentowana wartość kodu, a w drugim dwa bajty i to nawet wtedy, gdy drugi z nich (bardziej znaczący) repezentuje daną o wartości 0.

Przykład

```
inline(>$12/<$3456)
```

- Pierwszy element kodu reprezentuje daną jednobajtową, ale wobec użycia znaku > (większe) generuje dwa bajty: $12 i $00.
- Drugi element kodu reprezentuje daną dwubajtową, ale wobec użycia znaku < (mniejsze) generuje jeden bajt: $56.
- Przytoczona instrukcja inline generuje trzy bajty kodu: $12, $00 i $56, w podanej kolejności.
- Gdyby rozpatrywaną instrukcję zmieniono do postaci

```
inline($12/$3456)
```

to generowałaby ona trzy bajty: $12, $56 i $34. ■

Przykład (podano wg Turbo Pascal Reference Manual, wersja 3.0, rozdz. 22 - za zgodą Borland International)

```
type
   AnyString = string[255];
procedure ToUpper(var Str : AnyString);
begin
  inline(
    $2a/Str/
    $46/
    $04/
    $05/
    $ca/*+20/
```

```
        $23/
        $7e/
        $fe/$61/
        $da/*-9/
        $fe/$7b/
        $d2/*-14/
        $d6/$20/
        $77/
        $3c/*-20)
    end;
```

• Wykonanie przytoczonej procedury powoduje zamianę małych liter zmiennej łańcuchowej, z którą skojarzono parametr *Str*, na duże.

• Elementy kodu zawarte w instrukcji inline generują następujący podprogram asemblerowy

```
        LD  HL,(Str)
        LD  B,(HL)
        INC B
L1:     DEC B
        JP  Z,L2
        INC HL
        LD  A,(HL)
        CP  'a'
        JP  C,L1
        CP  'z'+1
        JP  NC,L1
        SUB 20H
        LD  (HL),A
        JP  L1
L2:
```

■

19.5. AKTYWOWANIE INNYCH PROGRAMOW

W programach zapisanych w języku Turbo Pascal można posługiwać się dwiema dodatkowymi procedurami umożliwiającymi aktywowanie innych programów. Wywołania tych procedur mają postać

Execute(FileVar)

oraz

Chain(FileVar)

gdzie *FileVar* jest nazwą zmiennej plikowej identyfikującej zbiór zawierający program, który ma zostać wykonany jako następny po programie właśnie wykonywanym.

W przypadku użycia procedury *Execute* program ten

powinien być zawarty w zbiorze z rozszerzeniem .COM, a w przypadku procedury *Chain* powinien być zawarty w zbiorze z rozszerzeniem .CHN.

Program wywołany za pośrednictwem procedury *Chain* może komunikować się z programem wywołującym za pomocą zmiennych globalnych albo za pomocą zmiennych znajdujących się pod ustalonymi adresami. Zmienne globalne, o ile zostaną wykorzystane, powinny zostać zadeklarowane jako pierwsze, zarówno w programie wywołującym, jak i wywoływanym, i to w takiej samej kolejności. Po spełnieniu tego wymagania odwołania do zmiennych występujące w obu programach dotyczą tych samych danych.

Przykład. Przedstawiono dwa programy, z których pierwszy przekazuje drugiemu daną łańcuchową oraz daną całkowitą

```
program First;
var
  LastName : string[8];
  Age : byte;
  NameFile : file;
begin
  Write('Jan');
  LastName := 'Bielecki';
  Age := 44;
  Assign(NameFile, 'Second.chn');
  Chain(NameFile)
end.
{*********************************}
program Second;
var
  SurName : string[8];
  Age : byte;
begin
  Writeln(SurName,' is',Age :3,'now')
end.
```

- Przyjęto, że program *Second* zostanie po skompilowaniu umieszczony w zbiorze *Second.chn*.
- Zmienna o wartości *'Bielecki'* jest w programie *First* reprezentowana przez nazwę *LastName*, a w programie *Second* przez nazwę *SurName*.
- Wykonanie obu programów spowoduje wyprowadzenie napisu

Jan Bielecki is 44 now ■

20.1. DANE TYPU PORZĄDKOWEGO

Dane typu porządkowego są reprezentowane w jednym albo w dwóch bajtach pamięci operacyjnej.

W jednym bajcie są reprezentowane dane typu *char*, dane typu wyliczeniowego związanego ze zbiorem liczącym nie więcej niż 256 elementów oraz dane typu okrojonego *min..max*, takiego, że zarówno *ord(min)*, jak i *ord(max)* należy do przedziału 0..255. W szczególności w jednym bajcie są więc reprezentowane dane typu *boolean* i *byte*.

W dwóch bajtach są reprezentowane dane typu *integer*, dane typu okrojonego, które nie mogą być reprezentowane w jednym bajcie, oraz dane typu wyliczeniowego o więcej niż 256 elementach. W przypadku danych zajmujących dwa bajty mniej znaczącym bajtem danych jest pierwszy z nich.

Przykład

```
type
   Color = (Red,Green,Blue,Yellow,Orange);
   Hue = Green..Yellow;
   Selector = (enum,card,bool);
   union = record
               case Selector of
                enum: (Rng : Hue);
                card: (Int : integer);
                bool: (Log : boolean)
               end;
const
   HueVar : union = (Rng : Green);
```

```
begin
  with HueVar do
    Write(Int,Log :5)
end.
```

● Ponieważ *ord(Green)*<=255 oraz *ord(Yellow)*<=255, pole *Rng* typu *Hue* jest reprezentowane w jednym bajcie.

● Mniej znaczący bajt pola *Int* pokrywa się z bajtem pola *Rng* i bajtem pola *Log*.

● Wykonanie instrukcji *Write* powoduje wyprowadzenie napisu

```
1 TRUE
```

■

20.2. DANE TYPU RZECZYWISTEGO

Dane typu *real* są reprezentowane w 6 bajtach pamięci operacyjnej.

Dane te składają się z jednobajtowego wykładnika i pięciobajtowej mantysy. Pod najniższym adresem pamięci występuje wykładnik, a za nim mantysa, w kolejności od bajtu najmniej znaczącego do najbardziej znaczącego. Wartość wykładnika jest reprezentowana z przesunięciem $80, a mantysa jest znormalizowana i pozbawiona najbardziej znaczącego bitu. Bit ten jest wykorzystany do reprezentowania jej znaku. Dana rzeczywista typu *real* o wartości 0.0 jest reprezentowana przez mantysę i wykładnik składające się z samych bitów 0.

Przykład

```
type
   union = record
             case boolean of
               false: (Float : real);
               true: (Arr : array[0..5] of byte)
           end;
const
   RealVar : union = (Arr : ($83,0,0,0,0,$80));
begin
   with RealVar do
     Write(Float)
end.
```

● Pierwszy bajt zmiennej *RealVar* jest daną bajtową o wartości $80, a więc reprezentuje wykładnik 3.

● Bajt o wartości $80 jest najbardziej znaczącym bajtem mantysy. Reprezentuje on znak - (minus) mantysy oraz mantysę

```
10000000 00000000 00000000 00000000
```

Najbardziej znaczący bit tego bajtu reprezentuje znak mantysy, tu - (minus).

● Ponieważ mantysa ma wartość -0.5, a wykładnik ma wartość 3, dana początkowa przypisana zmiennej *RealVar* ma wartość $-0.5 * 2^3 = -4.0$.

● Wykonanie instrukcji *Write* powoduje wyprowadzenie liczby -4.0. ■

20.3. DANE TYPU ŁAŃCUCHOWEGO

Dane typu **string**[*n*] są reprezentowane w *n*+1 bajtach pamięci.

Pierwszy bajt danej łańcuchowej określa liczbę znaków tej danej.

Przykład

```
const
   StrVar : string[5] = 'Janek';
procedure CutOff(var Par);
var
  Len : byte absolute Par;
begin
  Len := Len - 2
end;
begin
  Writeln(StrVar);
  CutOff(StrVar);
  Write(StrVar)
end.
```

● Operacja na zmiennej *Len* jest w istocie operacją na najbardziej znaczącym bajcie zmiennej *StrVar*.

● Wykonanie przytoczonego programu powoduje wyprowadzenie napisu

Janek
Jan ■

20.4. DANE TYPU MNOGOŚCIOWEGO

Dane typu mnogościowego, bazowane na typie porządkowym o najmniejszym elemencie *min* i największym elemencie *max* zajmują

ord(*max*) **div** 8 - *ord*(*min*) **div** 8 + 1

bajtów pamięci.

Dane mnogościowe są reprezentowane w taki sposób, jakby bazowy typ porządkowy składał się z 256 elementów. Ponieważ wymagałoby to zarezerwowania dla danej 32 bajtow pamięci, z których wiele nie byłoby w większości przypadków używanych, z tych 32 bajtów odrzuca się te bajty skrajne, których wartość nigdy nie ulega zmianie.

W szczególności dana typu **set of** 10..16 nie jest reprezentowana w postaci 32-bajtowej (literami x oznaczono aktywne pozycje danej)

00000000 ... 00000000 0000000x xxxxxx00 00000000

lecz w postaci 2-bajtowej

0000000x xxxxxx00

w kolejności od bajtu najmniej znaczącego do najbardziej znaczącego.

Przykład

```
type
  union = record
            case boolean of
              false: (aSet : set of 10..16);
              true: (anInt : integer)
          end;
const
  SetVar : union = (aSet :[16]);
begin
  with SetVar do
    Write(anInt)
end.
```

● Pole *aSet* zajmuje 2 bajty i jest reprezentowane w pamięci operacyjnej w postaci 00000000 00000001.

● Wykonanie instrukcji *Write* powoduje wyprowadzenie liczby 256. ■

20.5. DANE WSKAZUJĄCE

Dane wskazujące są reprezentowane w 2 bajtach pamięci. Każda z nich jest reprezentowana tak jak dana typu *integer* i określa adres pamięci operacyjnej. Dana reprezentowana przez **nil** składa się z samych bitów 0.

Przykład

```
type
  union = record
            case boolean of
```

```
                    false: (Ref : byte);
                    true: (Int : integer)
                end;
      const
        PtrVar : union = (Int : 0);
      begin
        Write(PtrVar^.Ref = nil)
      end.
```

● Zmienna *PtrVar* zajmuje 2 bajty.

● Wykonanie instrukcji *Write* powoduje wyprowadzenie napisu *TRUE*. ■

20.6. TABLICE

Elementy tablic są rozmieszczone w pamięci operacyjnej wierszami.

Przykład

```
      type
        union = record
                  case boolean of
                    false: (Arr : array[boolean,
                                        boolean] of byte);
                    true: (Vec : array[1..4] of byte)
                end;
      const
        ArrVar : union = (Vec : 11,12,21,22);
      var
        Index : 1..4;
      begin
        with ArrVar do
          for Index := 1 to 4 do
            Write(Vec[Index] :3)
      end.
```

● Wykonanie instrukcji *Write* powoduje wyprowadzenie liczb 11, 12, 21, 22. ■

20.7. DANE REKORDOWE

Pola rekordów są rozmieszczone w pamięci operacyjnej w kolejności ich wyszczególnienia w deklaracji. Jeśli rekord nie zawiera wariantów, to jego rozmiar jest równy sumie rozmiarów pól. W przeciwnym razie rozmiar rekordu jest równy sumie rozmiaru części stałej i rozmiaru tego z wariantów, który

wymaga najwięcej miejsca. Zarówno rekordy bez wariantów, jak i rekordy z wariantami są stałego rozmiaru. Informacja ta jest istotna z punktu widzenia operacji wejścia/wyjścia.

Przykład

```
var
  RecVar : record
             case Selector : integer of
               6: (Arr : array[1..3,1..2] of byte);
               8: (aSet : set of 'A'..'Z')
           end;
begin
  with RecVar do
    Write(SizeOf(RecVar),
          SizeOf(Arr) :2,
          SizeOf(aSet) :2)
end.
```

- Część stała rekordu *RecVar* ma rozmiar 2 bajty.
- Pierwszy wariant tego rekordu ma rozmiar 6 bajtów.
- Drugi wariant ma rozmiar 8 bajtów.
- Rekord ma rozmiar 10 bajtów.
- Wykonanie instrukcji *Write* powoduje wyprowadzenie liczb 10, 6 i 8. ■

LITERATURA

1. Bielecki J.: *Turbo Pascal z grafiką dla IBM PC*. WNT, Warszawa (w przygotowaniu).
2. Cherry G.: *Pascal Programming Structures*. Restone Publishing Company, Reston, Virginia 1980.
3. Iglewski M., Madey J., Matwin S.: *Pascal*. WNT, Warszawa 1986.
4. Jensen K., Wirth N.: *Pascal - User Manual and Report*. Springer Verlag, Berlin 1974.
5. Schauer H.: *Pascal fur Anfanger*. R. Oldenbourg Verlag, Wien 1976.
6. *Turbo Graphix Toolbox*. Borland International, Scotts Valley, California 1985.
7. *Turbo Pascal version* 3.0. Borland International, Scotts Valley, California 1985.
8. *Turbo Toolbox*. Borland International, Scoots Valley, California 1985.
9. *Turbo Tutor*. Borland International, Scoots Valley, California 1985.

Dec	Hex	Char		Dec	Hex	Char	
0	00	^@ NUL		27	1B	^[ESC	
1	01	^A SOH	*	28	1C	^\ FS	*
2	02	^B STX	*	29	1D	^] GS	*
3	03	^C ETX	*	30	1E	^^ RS	*
4	04	^D EOT	*	31	1F	^_ US	*
5	05	^E ENQ	*	32	20		
6	06	^F ACK	*	33	21	!	
7	07	^G BEL		34	22	"	
8	08	^H BS		35	23	#	
9	09	^I HT		36	24	$	
10	0A	^J LF		37	25	%	
11	0B	^K VT	*	38	26	&	
12	0C	^L FF	*	39	27	'	
13	0D	^M CR		40	28	(	
14	0E	^N SO		41	29	)	
15	0F	^O SI	*	42	2A	*	
16	10	^P DLE	*	43	2B	+	
17	11	^Q DC1	*	44	2C	,	
18	12	^R DC2	*	45	2D	-	
19	13	^S DC3	*	46	2E	.	
20	14	^T DC4		47	2F	/	
21	15	^U NAK	*	48	30	0	
22	16	^V SYN	*	49	31	1	
23	17	^W ETB	*	50	32	2	
24	18	^X CAN	*	51	33	3	
25	19	^Y EM	*	52	34	4	
26	1A	^Z SUB	*	53	35	5	

Dec	Hex	Char	Dec	Hex	Char
54	36	6	91	5B	[
55	37	7	92	5C	\
56	38	8	93	5D	]
57	39	9	94	5E	^
58	3A	:	95	5F	_
59	3B	;	96	60	' (μ)
60	3C	<	97	61	a
61	3D	=	98	62	b
62	3E	>	99	63	c
63	3F	?	100	64	d
64	40	@	101	65	e
65	41	A	102	66	f
66	42	B	103	67	g
67	43	C	104	68	h
68	44	D	105	69	i
69	45	E	106	6A	j
70	46	F	107	6B	k
71	47	G	108	6C	l
72	48	H	109	6D	m
73	49	I	110	6E	n
74	4A	J	111	6F	o
75	4B	K	112	70	p
76	4C	L	113	71	q
77	4D	M	114	72	r
78	4E	N	115	73	s
79	4F	O	116	74	t
80	50	P	117	75	u
81	51	Q	118	76	v
82	52	R	119	77	w
83	53	S	120	78	x
84	54	T	121	79	y
85	55	U	122	7A	z
86	56	V	123	7B	{
87	57	W	124	7C	¦
88	58	X	125	7D	}
89	59	Y	126	7E	~
90	5A	Z	127	7F	DEL (♦)

Uwaga. W systemie CP/J nie implementowano kodów oznaczonych symbolem *. Znaki ' i DEL zastąpiono znakami w nawiasach

B.1. BŁĘDY SYGNALIZOWANE PODCZAS KOMPILOWANIA PROGRAMOW (w terminologii projektantów Elwro 800 Jr)

1 Oczekiwany ';'
2 Oczekiwany ':'
3 Oczekiwany ','
4 Oczekiwany '('
5 Oczekiwany ')'
6 Oczekiwany '='
7 Oczekiwany ':='
8 Oczekiwany '['
9 Oczekiwany ']'
10 Oczekiwana '.'
11 Oczekiwane '..'
12 Oczekiwane 'BEGIN'
13 Oczekiwane 'DO'
14 Oczekiwane 'END'
15 Oczekiwane 'OF'
17 Oczekiwane 'THEN'
18 Oczekiwane 'TO' lub 'DOWNTO'
20 Oczekiwane wyrażenie boole'owskie
21 Oczekiwana zmienna plikowa
22 Oczekiwana stała całkowita
23 Oczekiwane wyrażenie całkowite
24 Oczekiwana zmienna całkowita
25 Oczekiwana stała całkowita lub rzeczywista
26 Oczekiwane wyrażenie całkowite lub rzeczywiste
27 Oczekiwana zmienna całkowita lub rzeczywista
28 Oczekiwana zmienna wskazującą
29 Oczekiwana zmienna rekordowa
30 Oczekiwany typ prosty

31 Oczekiwane wyrażenie proste
32 Oczekiwana stała łańcuchowa
33 Oczekiwane wyrażenie łańcuchowe
34 Oczekiwana zmienna łańcuchowa
35 Oczekiwany identyfikator pliku tekstowego
36 Oczekiwany identyfikator typu
37 Oczekiwany identyfikator pliku bez typu elementów
40 Niezdefiniowana etykieta
41 Nieznany identyfikator lub błąd składniowy
42 Nieznany typ wskazujący
43 Powtórzona etykieta lub identyfikator
44 Niewłaściwy typ
45 Wartość stałej poza zakresem
46 Błąd typu stałej lub wyróżnika CASE
47 Niewłaściwy argument lub operator
48 Niewłaściwy typ wyniku
49 Niewłaściwa długość łańcucha
50 Niewłaściwa długość stałej łańcuchowej
51 Niewłaściwy typ bazowy typu okrojonego
52 Górne ograniczenie mniejsze od dolnego
53 Słowo zarezerwowane
54 Niewłaściwe przypisanie
55 Stała łańcuchowa przekracza wiersz
56 Błąd w stałej całkowitej
57 Błąd w stałej rzeczywistej
58 Niewłaściwy znak w identyfikatorze
60 Stałe nie są tutaj dozwolone
61 Zmienne plikowe i wskazujące nie są tutaj dozwolone
62 Zmienne strukturalne nie są tutaj dozwolone
63 Pliki tekstowe nie są tutaj dozwolone
64 Pliki bez typu elementów i tekstowe - tutaj niedozwolone
65 Pliki bez typu elementów nie są tutaj dozwolone
66 Operacje we/wy nie są tutaj dozwolone
67 Zmienne plikowe wymagają deklaracji VAR
68 Pliki nie mogą być składnikami pliku
69 Niewłaściwe przyporządkowanie pól
70 Zbiór za duży
71 Niewłaściwe GOTO
72 Etykieta spoza bieżącego bloku
73 Niezdefiniowana procedura FORWARD
74 Błąd INLINE
75 Niewłaściwe użycie ABSOLUTE
90 Nieznana zmienna plikowa
91 Nieoczekiwany koniec programu
97 Za dużo zagnieżdżonych WITH

98 Przepełnienie pamięci
99 Przepełnienie kompilatora

B.2. BŁĘDY SYGNALIZOWANE PODCZAS WYKONYWANIA PROGRAMOW

Błędy, które mogą wystąpić podczas wykonywania programów dzielą się na *błędy fatalne* i *błędy operacji wejścia/wyjścia.*

Błędy fatalne są sygnalizowane za pomocą napisu

Run-time error *NN*, PC=*addr*

Program aborted

w którym *NN* jest numerem błędu, a *addr* jest adresem tego miejsca w programie, gdzie wykryto błąd.

Błędy operacji wejścia/wyjścia są sygnalizowane za pomocą napiu

I/O error *NN*, PC=*addr*
Program aborted

w którym *NN* i *addr* mają taką samą interpretację jak powyżej.

Błędy fatalne

01 Nadmiar stałopozycyjny.
02 Dzielenie przez zero.
03 Błąd argumentu funkcji *Sqrt*.
04 Błąd argumentu funkcji *Ln*.
10 Próba utworzenia danej łańcuchowej o rozmiarze przeczającym 255 znaków. Próba konwersji danej łańcuchowej, reprezentującej więcej niż jeden znak, w daną znakową.
90 Niewłaściwy indeks w odwołaniu do tablicy.
91 Próba przypisania danej spoza dopuszczalnego zakresu.
92 Wartość argumentu funkcji *Trunc* albo *Round* spoza dopuszczalnego zakresu.
FF Brak wolnego miejsca w pamięci.

Błędy operacji wejścia/wyjścia

01 Plik nie istnieje.
02 Plik nie jest otwarty.
03 Plik nie jest otwarty do wyprowadzania.
04 Plik nie jest otwarty.
10 Błąd reprezentacji liczby.
20 Zabroniona operacja.

21 Zabroniona operacja.
22 Zabronione użycie procedury *Assign*.
90 Niezgodność rozmiarów rekordów.
91 Próba wykroczenia poza plik.
99 Nieoczekiwany koniec pliku.
F0 Błąd operacji zapisu na dysk.
F1 Przepełnienie katalogu.
F2 Przepełnienie pliku.
FF Brak pliku.

C.1. POJĘCIA PODSTAWOWE

Edytor tekstów WordStar został wprowadzony na rynek mikrokomputerowy przez firmę MicroPro już w 1977 roku. Główną zaletą, która w znacznej mierze przyczyniła się do jego rozpowszechnienia, było założenie natychmiastowej edycji ekranowej oraz przyjęcie rozwiązań niezależnych od sprzętu niezbędnego do wdrożenia edytora. Mimo pojawienia się konkurencyjnych implementacji edytorów tekstów, WordStar zachował swoją wiodącą pozycję i w postaci kilku mutacji jest dostępny zarówno na mikrokomputerach 8-bitowych, jak i 16-bitowych. W dalszej części opisu zostanie on przedstawiony w postaci, w jakiej jest implementowany w systemie Turbo Pascal.

W odróżnieniu od wielu innych edytorów edytor WordStar jest zorientowany na wprowadzanie tekstu, a nie na przyjmowanie dyrektyw przetwarzania tekstu. Konsekwencją tego założenia jest sterowanie operacjami na tekście za pomocą znaków, które nie mają bezpośredniej reprezentacji graficznej. Takie znaki będą nazywane *znakami sterującymi*, a ich użycie będzie się sprowadzać do jednoczesnego naciśnięcia klawisza znaku CTRL oraz jednego albo dwóch klawiszy znaków literowych dodatkowych. Ponieważ w dalszym opisie odwoływanie się do znaków sterujących będzie występować bardzo często, zostanie przyjęta umowa oznaczania znaków sterujących za pomocą symbolu ^ reprezentującego naciśnięcie klawisza CTRL. W szczególności nadanie znaku sterującego CTRL A będzie w skrócie nazywane wydaniem dyrektywy ^A, co uzyskuje się przez naciśnięcie klawisza CTRL i jednoczesne naciśnięcie klawisza oznaczonego literą A. W tych przypadkach, gdy sterowanie edytorem odbywa się za pomocą pary znaków sterujących, np. ^KB (także nazywanej dyrektywą), należy nacisnąć klawisz

CTRL, a następnie kolejno klawisze oznaczone literami K i B (podczas naciskania drugiego z tych klawiszy zbędne jest wciskanie klawisza CTRL).

Typowa sesja współpracy z edytorem WordStar zaczyna się od wydania w głównym menu dyrektywy E. Powoduje to wywołanie edytora i umieszczenie na ekranie monitora początkowej części zbioru roboczego zidentyfikowanego uprzednio za pomocą dyrektywy W. Po wprowadzeniu tekstu tego zbioru albo po dokonaniu w nim zmian należy posłużyć się dyrektywą edytora ^KD. Spowoduje to zakończenie edycji i powrót do głównego menu. W tym momencie można posłużyć się dyrektywą S - w celu zapamiętania zbioru roboczego na dysku.

Bezpośrednio po wywołaniu edytora można traktować komputer jako wielofunkcyjną maszynę do pisania. Maszyna taka ma tę właściwość, że każdy wprowadzony znak pojawia się natychmiast na ekranie. Błędnie wprowadzone pojedyncze litery można usuwać za pomocą klawisza ← (Backspace). Jeśli błędy występują we wcześniej wprowadzonych słowach, to można przesunąć kursor nad takie słowa i dokonać natychmiastowej poprawki. Zasady takiego poprawiania zostaną omówione w następnym punkcie.

C.2. WPROWADZANIE, ZMIANY I USUWANIE PORCJI TEKSTU

Jak już wyjaśniono, bezpośrednio po aktywowaniu edytora następuje przejście do tworzenia albo aktualizowania tekstu. Odbywa się to przez wprowadzanie słów albo operowanie na słowach już wprowadzonych. Pod pojęciem słowa jest rozumiany dowolny ciąg znaków zakończony spacją albo jednym ze znaków przestankowych, np. , (przecinek), . (kropka), : (dwukropek), ' (apostrof). W sensie podanej definicji, przykładowy program

```
program JanB;
begin
  Write('I am JanB')
end.
```

składa się ze słów

```
program
JanB;
begin
Write('
I
am
JanB'
```

```
)
end.
```

W obrębie tekstu wyświetlanego na ekranie monitora jeden ze znaków jest wyróżniony za pomocą kursora. Operacje na znakach i słovach są zawsze wykonywane w odniesieniu do tego znaku, zawierającego go słowa albo wiersza. Przesuwanie kursora może także dotyczyć pojedynczych znaków, słów albo wierszy. Może ponadto dotyczyć ekranów, tj. porcji tekstu obejmujących większą liczbę wierszy, zbliżoną do pojemności ekranu monitora.

Przesunięcie kursora o jedną pozycję w lewo, w prawo, w górę i w dół odbywa się za pomocą dyrektyw ^E, ^S, ^D i ^X, których klawisze tworzą romb

```
  E
S   D
  X
```

Jak łatwo się domyślić, dyrektywa ^E przesuwa kursor o jeden wiersz w górę, dyrektywa ^X przesuwa go o jeden wiersz w dół, a dyrektywy ^S i ^D przesuwają kursor odpowiednio o jedną pozycję w lewo i o jedną pozycję w prawo. Dzięki takiemu rozwiązaniu o funkcji dyrektywy decyduje nie mnemonika nazwy jej klawisza, lecz położenie klawisza w przytoczonym układzie czterokierunkowym.

Przesuwanie kursora o pojedyncze słowa jest realizowane za pomocą dwóch dodatkowych dyrektyw ^A i ^F

```
    E
A S   D F
    X
```

Użycie dyrektywy ^A powoduje przesunięcie kursora o jedno słowo w lewo, a użycie dyrektywy ^F powoduje przesunięcie go o jedno słowo w prawo.

Kolejna para dyrektyw - ^R i ^C - służy do przesunięcia kursora odpowiednio do poprzedniego i następnego ekranu. Klawisze tych dyrektyw są usytuowane z prawej strony osi wyznaczonej przez klawisze E-X. Z lewej strony tej osi znajduje się para klawiszy, które przesuwają tekst na ekranie o jeden wiersz: ^W w dół i ^Z w górę. Tak więc ostatecznie układ klawiszy używanych do sterowania położeniem kursora oraz określających porcję wyświetlanego tekstu przybiera postać

```
W   E   R
A S   D F
Z   X   C
```

Układ ten jest łatwy do zapamiętania i nie wymaga identyfikowania klawiszy z opatrującymi je napisami.

Z prawej strony tego zestawu klawiszy znajdują się trzy klawisze dyrektyw ^T, ^Y, ^G, tworzące układ

T Y
G

Użycie dyrektywy ^G powoduje usunięcie znaku wyróżnionego przez kursor, użycie dyrektywy ^T powoduje usunięcie znaków od znaku wyróżnionego przez kursor do końca słowa, a użycie dyrektywy ^Y powoduje usunięcie wiersza, w którym znajduje się kursor.

Uzupełnieniem przytoczonego repertuaru dyrektyw jest znak sterujący ← (Backspace), którego naciśnięcie powoduje usunięcie znaku poprzedzającego znak wyróżniony przez kursor.

Podsumowanie

^E Przesunięcie kursora o jeden wiersz do góry, z zachowaniem pozycji kolumny, chyba że poprzedni wiersz jest krótszy, kiedy to kursor zajmie ostatnią pozycję wiersza.

^X Przesunięcie kursora o jeden wiersz do dołu z zachowaniem pozycji kolumny, chyba że następny wiersz jest krótszy, kiedy to kursor zajmie ostatnią pozycję wiersza.

^S Przesunięcie kursora o jedną pozycję w lewo. Jeśli przed przesunięciem kursor znajdował się w skrajnej lewej pozycji, to zostanie przesunięty na koniec poprzedniego wiersza.

^D Przesunięcie kursora o jedną pozycję w prawo. Jeśli przed przesunięciem kursor znajdował się w skrajnej prawej pozycji, to zostanie przesunięty na początek następnego wiersza.

^A Przesunięcie kursora do początku słowa w lewo.

^F Przesunięcie kursora do początku słowa w prawo.

^R Przesunięcie "okna", przez które jest widziany dokument, o jeden ekran do góry.

^C Przesunięcie "okna", przez które jest widziany dokument, o jeden ekran do dołu.

^W Przesunięcie tekstu na ekranie o jeden wiersz w dół.

^Z Przesunięcie tekstu na ekranie o jeden wiersz do góry.

^G Usunięcie znaku wyróżnionego przez kursor.

^T Usunięcie znaku wyróżnionego przez kursor oraz następujących po nim znaków danego słowa.

^Y Usunięcie wiersza, w którym znajduje się znak wyróżnio-

ny przez kursor.

← Usunięcie znaku poprzedzającego znak wyróżniony przez kursor.

Przytoczony tu repertuar dyrektyw, umożliwiający wizualizację i usuwanie dowolnych partii tekstu, został w edytorze WordStar wzbogacony dyrektywą ^N do wstawiania pustych wierszy oraz dodatkowym zestawem dyrektyw realizujących funkcje zbliżone do poprzednio omówionych, tyle że z większym "skokiem". Każda z takich dyrektyw składa się ze znaku ^Q, bezpośrednio po którym występuje jeden ze znaków tworzących układ

```
    E   R
  S   D
    X   C
```

Dyrektywa ^QE powoduje przesunięcie kursora do pierwszego wiersza ekranu, dyrektywa ^QX powoduje przesunięcie kursora do ostatniego wiersza ekranu, dyrektywa QS powoduje przesunięcie kursora do początku, a dyrektywa QD powoduje przesunięcie kursora do końca tego wiersza, w którym znajduje się kursor. Dwie pozostałe dyrektywy ^QR i ^QC powodują odpowiednio przesunięcie kursora do początku i do końca pliku.

Przesuwanie kursora za pomocą dowolnych zestawów wymienionych tu dyrektyw może służyć dwóm celom: przemieszczaniu się w obrębie pliku w celu sprawdzenia jego zawartości oraz przemieszczaniu się do obszaru, w którym mają być dokonane zmiany. W tym drugim przypadku zmiany mogą polegać na uzupełnieniu tekstu albo zastąpieniu go innym. W celu uzupełnienia tekstu wystarczy wprowadzić go z klawiatury. Edytor wstawi taki tekst przed znak wyróżniony przez kursor, dokonując na bieżąco przesunięcia znaku wyróżnionego oraz znaków po nim następujących. W celu zmiany tekstu można go najpierw usunąć, albo zastąpić nowym - na zasadzie uzupełnienia, albo stary tekst wyrugować nowym. To ostatnie rozwiązanie nie należy do najlepszych, gdyż rzadko zdarza się, aby stary i nowy tekst były tej samej długości. Ponadto jego zastosowanie wymaga przejściowego wyłączenia opcji rozsuwania tekstu, pociągając za sobą konieczność posłużenia się dyrektywą ^V. Każdorazowe użycie tego znaku powoduje przełączenie procesora ze stanu zastępowania tekstu w stan wstawiania tekstu i odwrotnie. Jego dokładne działanie najlepiej jest wypraktykować, śledząc zmiany zachodzące na ekranie.

C.3. ZŁOŻONE OPERACJE NA TEKŚCIE

Opisane dotychczas dyrektywy umożliwiają nie tylko utworzenie, lecz również praktycznie dowolne przetwarzanie dokumentów. Tym niemniej w pewnych przypadkach przydatne są operacje dotyczące większych fragmentów tekstu, takich jak przestawienie lub kopiowanie grup sąsiadujących wierszy albo wyszukiwanie lub zmiana określonych słów lub fraz. Do takich celów służą omówione poniżej dyrektywy blokowe.

Dyrektywa ^QF

Po wykonaniu tej dyrektywy na ekranie pojawia się napis

FIND:

stanowiący zapytanie o frazę, która ma być wyszukana w dokumencie. Fraza taka składa się z dowolnej porcji znaków, którą kończy naciśnięcie klawisza CR (Elwro 800 Jr) albo Enter (IBM PC), zwanego dalej RETURN. Bezpośrednio po naciśnięciu klawisza RETURN na ekranie pojawia się zapytanie

OPTIONS:

na które należy odpowiedzieć określeniem trybu poszukiwania frazy. W najczęściej używanym przypadku - poszukiwania frazy począwszy od pozycji wyróżnionej przez kursor w kierunku końca pliku - wystarczy w odpowiedzi na zapytanie nacisnąć klawisz RETURN. Jeśli poszukiwana fraza znajduje się w pliku, to kursor zostanie umieszczony bezpośrednio za ostatnim znakiem znalezionej frazy. Jak łatwo się domyślić, użycie dyrektywy ^QF nie jest niczym innym jak posłużeniem się dyrektywą edytora kontekstowego. Należy nadmienić, że wśród opcji określających tryb poszukiwania frazy znajdują się m.in. takie jak

B poszukuj wstecz,
W ogranicz się do pełnych słów,
U utożsamiaj litery duże i małe.

Ponadto można posłużyć się opcją mającą postać liczby. W takim przypadku, dla liczby *n*, poszukiwanie dotyczy *n*-tego wystąpienia poszukiwanej frazy.

Dyrektywa ^QA

Dyrektywa ^QA stanowi rozwinięcie dyrektywy ^QF. Bezpośrednio po jej użyciu także pojawia się zapytanie o poszukiwaną frazę, ale po naciśnięciu kończącego ją klawisza RETURN

pojawia się zapytanie o frazę

REPLACE WITH:

na które należy odpowiedzieć frazą, która ma zastąpić frazę podaną po FIND: . Po tej drugiej frazie, także zakończonej naciśnięciem klawisza RETURN, pojawia się znane pytanie o opcje określające tryb poszukiwania frazy. Oprócz opcji już wymienionych dopuszczlne są tu także opcje

G zastępuj w całym pliku (a więc nie od pozycji wyróżnionej przez kursor, lecz od pierwszego znaku pliku),

N zastępuj bezwarunkowo.

Jeśli użyto opcji N, to wykonanie dyrektywy przebiega aż do wykonania wszystkich zastąpień. Jeśli nie użyto tej opcji, to po każdym znalezieniu poszukiwanej frazy wyświetlane jest zapytanie, czy należy dokonać zastąpienia. Odpowiedź Y stanowi zgodę na zastąpienie. Dowolna inna odpowiedź powoduje zaniechanie proponowanego zastąpienia i podjęcie poszukiwania następnego wystąpienia zastępowanej frazy. Przerwanie tego procesu można uzyskać za pomocą dyrektywy ^U.

Dyrektywa ^KB

Wykonanie dyrektywy ^KB powoduje oznaczenie miejsca w tekście jako tzw. początku bloku. Zdefiniowanie bloku wymaga dodatkowo użycia dyrektywy ^KK.

Dyrektywa ^KK

Wykonanie dyrektywy ^KK powoduje oznaczenie miejsca w tekście jako tzw. końca bloku. Dyrektywa ta wraz z dyrektywą ^KB definiuje blok. Z chwilą zdefiniowania bloku ciąg znaków tekstu znajdujący się między pozycjami kursora w chwili wykonywania dyrektyw ^KB i ^KK zostaje wyróżniony za pomocą inwersji znaków albo ich przyciemnienia. Dalej opisane dyrektywy grupy ^Kx będą dotyczyć tej właśnie partii tekstu.

Dyrektywa ^KC

Wykonanie dyrektywy ^KC powoduje skopiowanie bloku wyróżnionego przez dyrektywy ^KB i ^KK na pozycję przed znak wyróżniony przez kursor. Po tej operacji kursor zostanie ustawiony na pierwszym znaku skopiowanego bloku.

Dyrektywa ^KV

Wykonanie dyrektywy ^KV powoduje przeniesienie bloku wyróżnionego przez dyrektywy ^KB i ^KK na pozycję przed znak wyróżniony przez kursor. Po tej operacji kursor zostanie ustawiony na pierwszym znaku przeniesionego bloku, który w swojej nowej pozycji pozostanie blokiem wyróżnionym.

Dyrektywa ^KY

Wykonanie dyrektywy ^KY powoduje usunięcie wyróżnionego bloku. Można dla przykładu nadmienić, że jedynym skutkiem wykonania ciągu ^KB, ^KK, ^KY jest anulowanie wyróżnienia bloku.

Dyrektywa ^KR

Wykonanie dyrektywy ^KR powoduje wstawienie w miejscu wyróżnionym przez kursor tekstu znajdującego się w zbiorze dyskowym. Bezpośrednio po nadaniu tej dyrektywy na ekranie monitora pojawia się zapytanie

Read block from file:

o nazwę włączanego zbioru.

Dyrektywa ^KW

Wykonanie dyrektywy ^KW powoduje wyprowadzenie do zbioru dyskowego bloku tekstu wyróżnionego przez dyrektywy ^KB i ^KK. Bezpośrednio po użyciu tej dyrektywy na ekranie monitora pojawia się zapytanie

Write block from file:

o nazwę zbioru, w którym zostanie umieszczony wyróżniony blok. Jeśli zbiór taki już istnieje, to przed umieszczeniem w nim bloku zostaje usunięty, a następnie utworzony ponownie.

C.4. DYREKTYWY POMOCNICZE

Dyrektywa ^N

Wykonanie dyrektywy ^N powoduje wstawienie w miejscu wyróżnionym przez kursor pary znaków cr-lf, tj. wstawienie nowego wiersza.

Dyrektywa ^I

Wykonanie dyrektywy ^I powoduje przesunięcie kursora do najbliższej pozycji tabulacyjnej. Pozycją tą jest początek słowa w wierszu ponad kursorem.

Dyrektywa ^L

Wykonanie dyrektywy ^L powoduje powtórzenie ostatniej dyrektywy ^QF.

Dyrektywa ^U

Wykonanie dyrektywy ^U powoduje zaniechanie wykonywania dowolnej dyrektywy.

Dyrektywa ^P

Wykonanie dyrektywy ^P powoduje, że następujący bezpośrednio za nią znak sterujący jest wprowadzany do pliku. W szczególności wprowadzenie sekwencji znaków ^P^M^P^J umożliwia identyfikację pary znaków cr-lf kończącej każdy wiersz tekstu.

Przykład
Przesunięcie całego tekstu o 5 kolumn w prawo wymaga wprowadzenia z klawiatury następujących znaków (znak RETURN oznaczono symbolem r, a spację symbolem s)

```
^QRsssss
^QA^P^M^P^Jr^P^M^P^JssssssrNGr
```

■

Cena zł 590,–

ISBN 83-204-0949-7